The Julia Language

The Julia Project

August 6, 2017

Contents

Part I

Home

Part II

Julia Documentation

Chapter 1

Manual

Chapter 2

Standard Library

Chapter 3

Developer Documentation

Part III

Manual

Chapter 4

Introduction

Scientific computing has traditionally required the highest performance, yet domain experts have largely moved to slower dynamic languages for daily work. We believe there are many good reasons to prefer dynamic languages for these applications, and we do not expect their use to diminish. Fortunately, modern language design and compiler techniques make it possible to mostly eliminate the performance trade-off and provide a single environment productive enough for prototyping and efficient enough for deploying performance-intensive applications. The Julia programming language fills this role: it is a flexible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically-typed languages.

Because Julia's compiler is different from the interpreters used for languages like Python or R, you may find that Julia's performance is unintuitive at first. If you find that something is slow, we highly recommend reading through the Performance Tips section before trying anything else. Once you understand how Julia works, it's easy to write code that's nearly as fast as C.

Julia features optional typing, multiple dispatch, and good performance, achieved using type inference and just-in-time (JIT) compilation, implemented using LLVM. It is multi-paradigm, combining features of imperative, functional, and object-oriented programming. Julia provides ease and expressiveness for high-level numerical computing, in the same way as languages such as R, MATLAB, and Python, but also supports general programming. To achieve this, Julia builds upon the lineage of mathematical programming languages, but also borrows much from popular dynamic languages, including Lisp, Perl, Python, Lua, and Ruby.

The most significant departures of Julia from typical dynamic languages are:

- The core language imposes very little; the standard library is written in Julia itself, including primitive operations like integer arithmetic
- A rich language of types for constructing and describing objects, that can also optionally be used to make type declarations
- The ability to define function behavior across many combinations of argument types via multiple dispatch
- Automatic generation of efficient, specialized code for different argument types
- Good performance, approaching that of statically-compiled languages like C

Although one sometimes speaks of dynamic languages as being "typeless", they are definitely not: every object, whether primitive or user-defined, has a type. The lack of type declarations in most dynamic languages, however, means that one cannot instruct the compiler about the types of values, and often cannot explicitly talk about types at all. In static languages, on the other hand, while one can – and usually must – annotate types for the compiler, types exist only at compile time and cannot be manipulated or expressed at run time. In Julia, types are themselves run-time objects, and can also be used to convey information to the compiler.

While the casual programmer need not explicitly use types or multiple dispatch, they are the core unifying features of Julia: functions are defined on different combinations of argument types, and applied by dispatching to the most specific matching definition. This model is a good fit for mathematical programming, where it is unnatural for the first argument to "own" an operation as in traditional object-oriented dispatch. Operators are just functions with special notation – to extend addition to new user-defined data types, you define new methods for the function. Existing code then seamlessly applies to the new data types.

Partly because of run-time type inference (augmented by optional type annotations), and partly because of a strong focus on performance from the inception of the project, Julia's computational efficiency exceeds that of other dynamic languages, and even rivals that of statically-compiled languages. For large scale numerical problems, speed always has been, continues to be, and probably always will be crucial: the amount of data being processed has easily kept pace with Moore's Law over the past decades.

Julia aims to create an unprecedented combination of ease-of-use, power, and efficiency in a single language. In addition to the above, some advantages of Julia over comparable systems include:

- Free and open source (MIT licensed)
- User-defined types are as fast and compact as built-ins
- No need to vectorize code for performance; devectorized code is fast
- Designed for parallelism and distributed computation
- Lightweight "green" threading (coroutines)
- Unobtrusive yet powerful type system
- Elegant and extensible conversions and promotions for numeric and other types
- Efficient support for Unicode, including but not limited to UTF-8
- Call C functions directly (no wrappers or special APIs needed)
- Powerful shell-like capabilities for managing other processes
- Lisp-like macros and other metaprogramming facilities

Chapter 5

Getting Started

Julia installation is straightforward, whether using precompiled binaries or compiling from source. Download and install Julia by following the instructions at https://julialang.org/downloads/.

The easiest way to learn and experiment with Julia is by starting an interactive session (also known as a read-eval-print loop or "repl") by double-clicking the Julia executable or running from the command line:

To exit the interactive session, type – the control key together with the key or type . When run in interactive mode, displays a banner and prompts the user for input. Once the user has entered a complete expression, such as , and hits enter, the interactive session evaluates the expression and shows its value. If an expression is entered into an interactive session with a trailing semicolon, its value is not shown. The variable is bound to the value of the last evaluated expression whether it is shown or not. The variable is only bound in interactive sessions, not when Julia code is run in other ways.

To evaluate expressions written in a source file , write .

To run code in a file non-interactively, you can give it as the first argument to the command:

As the example implies, the following command-line arguments to are taken as command-line arguments to the program , passed in the global constant . The name of the script itself is passed in as the global . Note that is also set when script code is given using the option on the command line (see the help output below) but will be empty. For example, to just print the arguments given to a script, you could do this:

Or you could put that code into a script and run it:

The delimiter can be used to separate command-line args to the scriptfile from args to Julia:

Julia can be started in parallel mode with either the or the options. will launch an additional worker processes, while will launch a worker for each line in file . The machines defined in must be accessible via a passwordless login, with Julia installed at the same location as the current host. Each machine definition takes the form . defaults to current user, to the standard ssh port. is the number of workers to spawn on the node, and defaults to 1. The optional specifies the ip-address and port that other workers should use to connect to this worker.

If you have code that you want executed whenever Julia is run, you can put it in :

There are various ways to run Julia code and provide options, similar to those available for the and programs:

5.1 Resources

In addition to this manual, there are various other resources that may help new users get started with Julia:

- Julia and IJulia cheatsheet
- Learn Julia in a few minutes
- Learn Julia the Hard Way
- Julia by Example
- Hands-on Julia
- Tutorial for Homer Reid's numerical analysis class
- An introductory presentation
- Videos from the Julia tutorial at MIT
- YouTube videos from the JuliaCons

Chapter 6

Variables

A variable, in Julia, is a name associated (or bound) to a value. It's useful when you want to store a value (that you obtained after some math, for example) for later use. For example:

Julia provides an extremely flexible system for naming variables. Variable names are case-sensitive, and have no semantic meaning (that is, the language will not treat variables differently based on their names).

Unicode names (in UTF-8 encoding) are allowed:

In the Julia REPL and several other Julia editing environments, you can type many Unicode math symbols by typing the backslashed LaTeX symbol name followed by tab. For example, the variable name can be entered by typing *-tab*, or even by *-tab-- tab--tab*. (If you find a symbol somewhere, e.g. in someone else's code, that you don't know how to type, the REPL help will tell you: just type and then paste the symbol.)

Julia will even let you redefine built-in constants and functions if needed:

However, this is obviously not recommended to avoid potential confusion.

6.1 Allowed Variable Names

Variable names must begin with a letter (A-Z or a-z), underscore, or a subset of Unicode code points greater than 00A0; in particular, Unicode character categories Lu/Ll/Lt/Lm/Lo/Nl (letters), Sc/So (currency and other symbols), and a few other letter-like characters (e.g. a subset of the Sm math symbols) are allowed. Subsequent characters may also include ! and digits (0-9 and other characters in categories Nd/No), as well as other Unicode code points: diacritics and other modifying marks (categories Mn/Mc/Me/Sk), some punctuation connectors (category Pc), primes, and a few other characters.

Operators like are also valid identifiers, but are parsed specially. In some contexts, operators can be used just like variables; for example refers to the addition function, and will reassign it. Most of the Unicode infix operators (in category Sm), such as , are parsed as infix operators and are available for user-defined methods (e.g. you can use to define as an infix Kronecker product).

The only explicitly disallowed names for variables are the names of built-in statements:

Some Unicode characters are considered to be equivalent in identifiers. Different ways of entering Unicode combining characters (e.g., accents) are treated as equivalent (specifically, Julia identifiers are NFC-normalized). The Unicode characters (U+025B: Latin small letter open e) and (U+00B5: micro sign) are treated as equivalent to the corresponding Greek letters, because the former are easily accessible via some input methods.

6.2 Stylistic Conventions

While Julia imposes few restrictions on valid names, it has become useful to adopt the following conventions:

- Names of variables are in lower case.
- Word separation can be indicated by underscores (), but use of underscores is discouraged unless the name would be hard to read otherwise.
- Names of s and s begin with a capital letter and word separation is shown with upper camel case instead of underscores.
- Names of s and s are in lower case, without underscores.
- Functions that write to their arguments have names that end in . These are sometimes called ”mutating” or ”in-place” functions because they are intended to produce changes in their arguments after the function is called, not just return a value.

For more information about stylistic conventions, see the Style Guide.

Chapter 7

Integers and Floating-Point Numbers

Integers and floating-point values are the basic building blocks of arithmetic and computation. Built-in representations of such values are called numeric primitives, while representations of integers and floating-point numbers as immediate values in code are known as numeric literals. For example, is an integer literal, while is a floating-point literal; their binary in-memory representations as objects are numeric primitives.

Julia provides a broad range of primitive numeric types, and a full complement of arithmetic and bitwise operators as well as standard mathematical functions are defined over them. These map directly onto numeric types and operations that are natively supported on modern computers, thus allowing Julia to take full advantage of computational resources. Additionally, Julia provides software support for Arbitrary Precision Arithmetic, which can handle operations on numeric values that cannot be represented effectively in native hardware representations, but at the cost of relatively slower performance.

The following are Julia's primitive numeric types:

- **Integer types:**

Type	Signed?	Number of bits	Smallest value	Largest value
		8	-2^7	2^7 - 1
		8	0	2^8 - 1
		16	-2^15	2^15 - 1
		16	0	2^16 - 1
		32	-2^31	2^31 - 1
		32	0	2^32 - 1
		64	-2^63	2^63 - 1
		64	0	2^64 - 1
		128	-2^127	2^127 - 1
		128	0	2^128 - 1
	N/A	8	(0)	(1)

- **Floating-point types:**

Additionally, full support for Complex and Rational Numbers is built on top of these primitive numeric types. All numeric types interoperate naturally without explicit casting, thanks to a flexible, user-extensible type promotion system.

Type	Precision	Number of bits
	half	16
	single	32
	double	64

7.1 Integers

Literal integers are represented in the standard manner:

The default type for an integer literal depends on whether the target system has a 32-bit architecture or a 64-bit architecture:

The Julia internal variable indicates whether the target system is 32-bit or 64-bit:

Julia also defines the types and , which are aliases for the system's signed and unsigned native integer types respectively:

Larger integer literals that cannot be represented using only 32 bits but can be represented in 64 bits always create 64-bit integers, regardless of the system type:

Unsigned integers are input and output using the prefix and hexadecimal (base 16) digits (the capitalized digits also work for input). The size of the unsigned value is determined by the number of hex digits used:

This behavior is based on the observation that when one uses unsigned hex literals for integer values, one typically is using them to represent a fixed numeric byte sequence, rather than just an integer value.

Recall that the variable is set to the value of the last expression evaluated in an interactive session. This does not occur when Julia code is run in other ways.

Binary and octal literals are also supported:

The minimum and maximum representable values of primitive numeric types such as integers are given by the and functions:

The values returned by and are always of the given argument type. (The above expression uses several features we have yet to introduce, including for loops, Strings, and Interpolation, but should be easy enough to understand for users with some existing programming experience.)

Overflow behavior

In Julia, exceeding the maximum representable value of a given type results in a wraparound behavior:

Thus, arithmetic with Julia integers is actually a form of modular arithmetic. This reflects the characteristics of the underlying arithmetic of integers as implemented on modern computers. In applications where overflow is possible, explicit checking for wraparound produced by overflow is essential; otherwise, the type in Arbitrary Precision Arithmetic is recommended instead.

Division errors

Integer division (the function) has two exceptional cases: dividing by zero, and dividing the lowest negative number () by -1. Both of these cases throw a . The remainder and modulus functions (and) throw a when their second argument is zero.

7.2 Floating-Point Numbers

Literal floating-point numbers are represented in the standard formats:

The above results are all values. Literal values can be entered by writing an in place of :

Values can be converted to easily:

Hexadecimal floating-point literals are also valid, but only as values:

Half-precision floating-point numbers are also supported (), but they are implemented in software and use for calculations.

The underscore can be used as digit separator:

Floating-point zero

Floating-point numbers have two zeros, positive zero and negative zero. They are equal to each other but have different binary representations, as can be seen using the function: :

Special floating-point values

There are three specified standard floating-point values that do not correspond to any point on the real number line:

			Name	Description
			positive infinity	a value greater than all finite floating-point values
			negative infinity	a value less than all finite floating-point values
			not a number	a value not to any floating-point value (including itself)

For further discussion of how these non-finite floating-point values are ordered with respect to each other and other floats, see Numeric Comparisons. By the IEEE 754 standard, these floating-point values are the results of certain arithmetic operations:

The and functions also apply to floating-point types:

Machine epsilon

Most real numbers cannot be represented exactly with floating-point numbers, and so for many purposes it is important to know the distance between two adjacent representable floating-point numbers, which is often known as machine epsilon.

Julia provides , which gives the distance between and the next larger representable floating-point value:

These values are and as and values, respectively. The function can also take a floating-point value as an argument, and gives the absolute difference between that value and the next representable floating point value. That is, yields a value of the same type as such that is the next representable floating-point value larger than :

The distance between two adjacent representable floating-point numbers is not constant, but is smaller for smaller values and larger for larger values. In other words, the representable floating-point numbers are densest in the real number line near zero, and grow sparser exponentially as one moves farther away from zero. By definition, is the same as since is a 64-bit floating-point value.

Julia also provides the and functions which return the next largest or smallest representable floating-point number to the argument respectively:

This example highlights the general principle that the adjacent representable floating-point numbers also have adjacent binary integer representations.

Rounding modes

If a number doesn't have an exact floating-point representation, it must be rounded to an appropriate representable value, however, if wanted, the manner in which this rounding is done can be changed according to the rounding modes presented in the IEEE 754 standard.

The default mode used is always , which rounds to the nearest representable value, with ties rounded towards the nearest value with an even least significant bit.

> **Warning**
>
> Rounding is generally only correct for basic arithmetic functions (, , , and) and type conversion operations. Many other functions assume the default mode is set, and can give erroneous results when operating under other rounding modes.

Background and References

Floating-point arithmetic entails many subtleties which can be surprising to users who are unfamiliar with the low-level implementation details. However, these subtleties are described in detail in most books on scientific computation, and also in the following references:

- The definitive guide to floating point arithmetic is the IEEE 754-2008 Standard; however, it is not available for free online.
- For a brief but lucid presentation of how floating-point numbers are represented, see John D. Cook's article on the subject as well as his introduction to some of the issues arising from how this representation differs in behavior from the idealized abstraction of real numbers.
- Also recommended is Bruce Dawson's series of blog posts on floating-point numbers.
- For an excellent, in-depth discussion of floating-point numbers and issues of numerical accuracy encountered when computing with them, see David Goldberg's paper What Every Computer Scientist Should Know About Floating-Point Arithmetic.
- For even more extensive documentation of the history of, rationale for, and issues with floating-point numbers, as well as discussion of many other topics in numerical computing, see the collected writings of William Kahan, commonly known as the "Father of Floating-Point". Of particular interest may be An Interview with the Old Man of Floating-Point.

7.3 Arbitrary Precision Arithmetic

To allow computations with arbitrary-precision integers and floating point numbers, Julia wraps the GNU Multiple Precision Arithmetic Library (GMP) and the GNU MPFR Library, respectively. The and types are available in Julia for arbitrary precision integer and floating point numbers respectively.

Constructors exist to create these types from primitive numerical types, and can be used to construct them from s. Once created, they participate in arithmetic with all other numeric types thanks to Julia's type promotion and conversion mechanism:

However, type promotion between the primitive types above and / is not automatic and must be explicitly stated.

The default precision (in number of bits of the significand) and rounding mode of operations can be changed globally by calling and , and all further calculations will take these changes in account. Alternatively, the precision or the rounding can be changed only within the execution of a particular block of code by using the same functions with a block:

7.4 Numeric Literal Coefficients

To make common numeric formulas and expressions clearer, Julia allows variables to be immediately preceded by a numeric literal, implying multiplication. This makes writing polynomial expressions much cleaner:

It also makes writing exponential functions more elegant:

The precedence of numeric literal coefficients is the same as that of unary operators such as negation. So is parsed as , and is parsed as .

Numeric literals also work as coefficients to parenthesized expressions:

Additionally, parenthesized expressions can be used as coefficients to variables, implying multiplication of the expression by the variable:

Neither juxtaposition of two parenthesized expressions, nor placing a variable before a parenthesized expression, however, can be used to imply multiplication:

Both expressions are interpreted as function application: any expression that is not a numeric literal, when immediately followed by a parenthetical, is interpreted as a function applied to the values in parentheses (see Functions for more about functions). Thus, in both of these cases, an error occurs since the left-hand value is not a function.

The above syntactic enhancements significantly reduce the visual noise incurred when writing common mathematical formulae. Note that no whitespace may come between a numeric literal coefficient and the identifier or parenthesized expression which it multiplies.

Syntax Conflicts

Juxtaposed literal coefficient syntax may conflict with two numeric literal syntaxes: hexadecimal integer literals and engineering notation for floating-point literals. Here are some situations where syntactic conflicts arise:

- The hexadecimal integer literal expression could be interpreted as the numeric literal multiplied by the variable .

- The floating-point literal expression could be interpreted as the numeric literal multiplied by the variable , and similarly with the equivalent form.

In both cases, we resolve the ambiguity in favor of interpretation as a numeric literals:

- Expressions starting with are always hexadecimal literals.
- Expressions starting with a numeric literal followed by or are always floating-point literals.

7.5 Literal zero and one

Julia provides functions which return literal 0 and 1 corresponding to a specified type or the type of a given variable.

Function	Description
	Literal zero of type or type of variable
	Literal one of type or type of variable

These functions are useful in Numeric Comparisons to avoid overhead from unnecessary type conversion.

Examples:

Chapter 8

Mathematical Operations and Elementary Functions

Julia provides a complete collection of basic arithmetic and bitwise operators across all of its numeric primitive types, as well as providing portable, efficient implementations of a comprehensive collection of standard mathematical functions.

8.1 Arithmetic Operators

The following arithmetic operators are supported on all primitive numeric types:

Expression	Name	Description
	unary plus	the identity operation
	unary minus	maps values to their additive inverses
	binary plus	performs addition
	binary minus	performs subtraction
	times	performs multiplication
	divide	performs division
	inverse divide	equivalent to
	power	raises to the th power
	remainder	equivalent to

as well as the negation on types:

Expression	Name	Description
	negation	changes to and vice versa

Julia's promotion system makes arithmetic operations on mixtures of argument types "just work" naturally and automatically. See Conversion and Promotion for details of the promotion system.

Here are some simple examples using arithmetic operators:

(By convention, we tend to space operators more tightly if they get applied before other nearby operators. For instance, we would generally write to reflect that first gets negated, and then is added to that result.)

8.2 Bitwise Operators

The following bitwise operators are supported on all primitive integer types:

Expression	Name
	bitwise not
	bitwise and
	bitwise or
	bitwise xor (exclusive or)
	logical shift right
	arithmetic shift right
	logical/arithmetic shift left

Here are some examples with bitwise operators:

8.3 Updating operators

Every binary arithmetic and bitwise operator also has an updating version that assigns the result of the operation back into its left operand. The updating version of the binary operator is formed by placing a immediately after the operator. For example, writing is equivalent to writing :

The updating versions of all the binary arithmetic and bitwise operators are:

Note

An updating operator rebinds the variable on the left-hand side. As a result, the type of the variable may change.

8.4 Vectorized "dot" operators

For *every* binary operation like , there is a corresponding "dot" operation that is *automatically* defined to perform element-by-element on arrays. For example, is not defined, since there is no standard mathematical meaning to "cubing" an array, but is defined as computing the elementwise (or "vectorized") result . Similarly for unary operators like or , there is a corresponding that applies the operator elementwise.

More specifically, is parsed as the "dot" call , which performs a broadcast operation: it can combine arrays and scalars, arrays of the same size (performing the operation elementwise), and even arrays of different shapes (e.g. combining row and column vectors to produce a matrix). Moreover, like all vectorized "dot calls," these "dot operators" are *fusing*. For example, if you compute (or equivalently , using the macro) for an array , it performs a *single* loop over , computing for each element of . In particular, nested dot calls like are fused, and "adjacent" binary operators like are equivalent to nested dot calls .

Furthermore, "dotted" updating operators like (or) are parsed as , where is a fused *in-place* assignment operation (see the dot syntax documentation).

Note the dot syntax is also applicable to user-defined operators. For example, if you define to give a convenient infix syntax for Kronecker products (), then will compute with no additional coding.

Combining dot operators with numeric literals can be ambiguous. For example, it is not clear whether means or . Therefore this syntax is disallowed, and spaces must be used around the operator in such cases.

8.5 Numeric Comparisons

Standard comparison operations are defined for all the primitive numeric types:

Here are some simple examples:

Operator	Name
	equality
,	inequality
	less than
,	less than or equal to
	greater than
,	greater than or equal to

Integers are compared in the standard manner – by comparison of bits. Floating-point numbers are compared according to the IEEE 754 standard:

- Finite numbers are ordered in the usual manner.
- Positive zero is equal but not greater than negative zero.
- is equal to itself and greater than everything else except .
- is equal to itself and less then everything else except .
- is not equal to, not less than, and not greater than anything, including itself.

The last point is potentially surprising and thus worth noting:

and can cause especial headaches with Arrays:

Julia provides additional functions to test numbers for special values, which can be useful in situations like hash key comparisons:

Function	Tests if
	and are identical
	is a finite number
	is infinite
	is not a number

considers s equal to each other:

can also be used to distinguish signed zeros:

Mixed-type comparisons between signed integers, unsigned integers, and floats can be tricky. A great deal of care has been taken to ensure that Julia does them correctly.

For other types, defaults to calling , so if you want to define equality for your own types then you only need to add a method. If you define your own equality function, you should probably define a corresponding method to ensure that implies .

Chaining comparisons

Unlike most languages, with the notable exception of Python, comparisons can be arbitrarily chained:

Chaining comparisons is often quite convenient in numerical code. Chained comparisons use the operator for scalar comparisons, and the operator for elementwise comparisons, which allows them to work on arrays. For example, gives a boolean array whose entries are true where the corresponding elements of are between 0 and 1.

Note the evaluation behavior of chained comparisons:

The middle expression is only evaluated once, rather than twice as it would be if the expression were written as . However, the order of evaluations in a chained comparison is undefined. It is strongly recommended not to use expressions with side effects (such as printing) in chained comparisons. If side effects are required, the short-circuit operator should be used explicitly (see Short-Circuit Evaluation).

Elementary Functions

Julia provides a comprehensive collection of mathematical functions and operators. These mathematical operations are defined over as broad a class of numerical values as permit sensible definitions, including integers, floating-point numbers, rationals, and complex numbers, wherever such definitions make sense.

Moreover, these functions (like any Julia function) can be applied in "vectorized" fashion to arrays and other collections with the dot syntax , e.g. will compute the sine of each element of an array .

8.6 Operator Precedence

Julia applies the following order of operations, from highest precedence to lowest:

For a complete list of *every* Julia operator's precedence, see the top of this file:

You can also find the numerical precedence for any given operator via the built-in function , where higher numbers take precedence:

Category	Operators
Syntax	followed by
Exponentiation	
Fractions	
Multiplication	
Bitshifts	
Addition	
Syntax	followed by
Comparisons	
Control flow	followed by followed by
Assignments	

8.7 Numerical Conversions

Julia supports three forms of numerical conversion, which differ in their handling of inexact conversions.

- The notation or converts to a value of type .
  - If is a floating-point type, the result is the nearest representable value, which could be positive or negative infinity.
  - If is an integer type, an is raised if is not representable by .
- converts an integer to a value of integer type congruent to modulo , where is the number of bits in . In other words, the binary representation is truncated to fit.
- The Rounding functions take a type as an optional argument. For example, is a shorthand for .

The following examples show the different forms.

See Conversion and Promotion for how to define your own conversions and promotions.

Rounding functions

Function	Description	Return type
	round to the nearest integer	
	round to the nearest integer	
	round towards	
	round towards	
	round towards	
	round towards	
	round towards zero	
	round towards zero	

Division functions

Function	Description
	truncated division; quotient rounded towards zero
	floored division; quotient rounded towards
	ceiling division; quotient rounded towards
	remainder; satisfies ; sign matches
	modulus; satisfies ; sign matches
	with offset 1; returns for or for , where
	modulus with respect to 2pi;
	returns
	returns
	greatest positive common divisor of , ,...
	least positive common multiple of , ,...

Function	Description
	a positive value with the magnitude of
	the squared magnitude of
	indicates the sign of , returning -1, 0, or +1
	indicates whether the sign bit is on (true) or off (false)
	a value with the magnitude of and the sign of
	a value with the magnitude of and the sign of

Function	Description
,	square root of
,	cube root of
	hypotenuse of right-angled triangle with other sides of length and
	natural exponential function at
	accurate for near zero
	computed efficiently for integer values of
	natural logarithm of
	base logarithm of
	base 2 logarithm of
	base 10 logarithm of
	accurate for near zero
	binary exponent of
	binary significand (a.k.a. mantissa) of a floating-point number

Sign and absolute value functions

Powers, logs and roots

For an overview of why functions like , , and are necessary and useful, see John D. Cook's excellent pair of blog posts on the subject: expm1, log1p, erfc, and hypot.

Trigonometric and hyperbolic functions

All the standard trigonometric and hyperbolic functions are also defined:

These are all single-argument functions, with the exception of atan2, which gives the angle in radians between the *x*-axis and the point specified by its arguments, interpreted as *x* and *y* coordinates.

Additionally, and are provided for more accurate computations of and respectively.

In order to compute trigonometric functions with degrees instead of radians, suffix the function with . For example, computes the sine of where is specified in degrees. The complete list of trigonometric functions with degree variants is:

Special functions

Function	Description
	gamma function at
	accurate for large
	accurate for large ; same as for , zero otherwise
	beta function at
	accurate for large or

Chapter 9

Complex and Rational Numbers

Julia ships with predefined types representing both complex and rational numbers, and supports all standard Mathematical Operations and Elementary Functions on them. Conversion and Promotion are defined so that operations on any combination of predefined numeric types, whether primitive or composite, behave as expected.

9.1 Complex Numbers

The global constant is bound to the complex number *i*, representing the principal square root of -1. It was deemed harmful to co-opt the name for a global constant, since it is such a popular index variable name. Since Julia allows numeric literals to be juxtaposed with identifiers as coefficients, this binding suffices to provide convenient syntax for complex numbers, similar to the traditional mathematical notation:

You can perform all the standard arithmetic operations with complex numbers:

The promotion mechanism ensures that combinations of operands of different types just work:

Note that , since a literal coefficient binds more tightly than division.

Standard functions to manipulate complex values are provided:

As usual, the absolute value () of a complex number is its distance from zero. gives the square of the absolute value, and is of particular use for complex numbers where it avoids taking a square root. returns the phase angle in radians (also known as the *argument* or *arg* function). The full gamut of other Elementary Functions is also defined for complex numbers:

Note that mathematical functions typically return real values when applied to real numbers and complex values when applied to complex numbers. For example, behaves differently when applied to versus even though :

The literal numeric coefficient notation does not work when constructing a complex number from variables. Instead, the multiplication must be explicitly written out:

However, this is *not* recommended; Use the function instead to construct a complex value directly from its real and imaginary parts:

This construction avoids the multiplication and addition operations.

and propagate through complex numbers in the real and imaginary parts of a complex number as described in the Special floating-point values section:

9.2 Rational Numbers

Julia has a rational number type to represent exact ratios of integers. Rationals are constructed using the operator:

If the numerator and denominator of a rational have common factors, they are reduced to lowest terms such that the denominator is non-negative:

This normalized form for a ratio of integers is unique, so equality of rational values can be tested by checking for equality of the numerator and denominator. The standardized numerator and denominator of a rational value can be extracted using the and functions:

Direct comparison of the numerator and denominator is generally not necessary, since the standard arithmetic and comparison operations are defined for rational values:

Rationals can be easily converted to floating-point numbers:

Conversion from rational to floating-point respects the following identity for any integral values of and , with the exception of the case and :

Constructing infinite rational values is acceptable:

Trying to construct a rational value, however, is not:

As usual, the promotion system makes interactions with other numeric types effortless:

Chapter 10

Strings

Strings are finite sequences of characters. Of course, the real trouble comes when one asks what a character is. The characters that English speakers are familiar with are the letters , , , etc., together with numerals and common punctuation symbols. These characters are standardized together with a mapping to integer values between 0 and 127 by the ASCII standard. There are, of course, many other characters used in non-English languages, including variants of the ASCII characters with accents and other modifications, related scripts such as Cyrillic and Greek, and scripts completely unrelated to ASCII and English, including Arabic, Chinese, Hebrew, Hindi, Japanese, and Korean. The Unicode standard tackles the complexities of what exactly a character is, and is generally accepted as the definitive standard addressing this problem. Depending on your needs, you can either ignore these complexities entirely and just pretend that only ASCII characters exist, or you can write code that can handle any of the characters or encodings that one may encounter when handling non-ASCII text. Julia makes dealing with plain ASCII text simple and efficient, and handling Unicode is as simple and efficient as possible. In particular, you can write C-style string code to process ASCII strings, and they will work as expected, both in terms of performance and semantics. If such code encounters non-ASCII text, it will gracefully fail with a clear error message, rather than silently introducing corrupt results. When this happens, modifying the code to handle non-ASCII data is straightforward.

There are a few noteworthy high-level features about Julia's strings:

- The built-in concrete type used for strings (and string literals) in Julia is . This supports the full range of Unicode characters via the UTF-8 encoding. (A function is provided to convert to/from other Unicode encodings.)
- All string types are subtypes of the abstract type , and external packages define additional subtypes (e.g. for other encodings). If you define a function expecting a string argument, you should declare the type as in order to accept any string type.
- Like C and Java, but unlike most dynamic languages, Julia has a first-class type representing a single character, called . This is just a special kind of 32-bit primitive type whose numeric value represents a Unicode code point.
- As in Java, strings are immutable: the value of an object cannot be changed. To construct a different string value, you construct a new string from parts of other strings.
- Conceptually, a string is a *partial function* from indices to characters: for some index values, no character value is returned, and instead an exception is thrown. This allows for efficient indexing into strings by the byte index of an encoded representation rather than by a character index, which cannot be implemented both efficiently and simply for variable-width encodings of Unicode strings.

10.1 Characters

A value represents a single character: it is just a 32-bit primitive type with a special literal representation and appropriate arithmetic behaviors, whose numeric value is interpreted as a Unicode code point. Here is how values are input

and shown:

You can convert a to its integer value, i.e. code point, easily:

On 32-bit architectures, will be . You can convert an integer value back to a just as easily:

Not all integer values are valid Unicode code points, but for performance, the conversion does not check that every character value is valid. If you want to check that each converted value is a valid code point, use the function:

As of this writing, the valid Unicode code points are through and through . These have not all been assigned intelligible meanings yet, nor are they necessarily interpretable by applications, but all of these values are considered to be valid Unicode characters.

You can input any Unicode character in single quotes using followed by up to four hexadecimal digits or followed by up to eight hexadecimal digits (the longest valid value only requires six):

Julia uses your system's locale and language settings to determine which characters can be printed as-is and which must be output using the generic, escaped or input forms. In addition to these Unicode escape forms, all of C's traditional escaped input forms can also be used:

You can do comparisons and a limited amount of arithmetic with values:

10.2 String Basics

String literals are delimited by double quotes or triple double quotes:

If you want to extract a character from a string, you index into it:

All indexing in Julia is 1-based: the first element of any integer-indexed object is found at index 1. (As we will see below, this does not necessarily mean that the last element is found at index , where is the length of the string.)

In any indexing expression, the keyword can be used as a shorthand for the last index (computed by). You can perform arithmetic and other operations with , just like a normal value:

Using an index less than 1 or greater than raises an error:

You can also extract a substring using range indexing:

Notice that the expressions and do not give the same result:

The former is a single character value of type , while the latter is a string value that happens to contain only a single character. In Julia these are very different things.

10.3 Unicode and UTF-8

Julia fully supports Unicode characters and strings. As discussed above, in character literals, Unicode code points can be represented using Unicode and escape sequences, as well as all the standard C escape sequences. These can likewise be used to write string literals:

Whether these Unicode characters are displayed as escapes or shown as special characters depends on your terminal's locale settings and its support for Unicode. String literals are encoded using the UTF-8 encoding. UTF-8 is a variable-width encoding, meaning that not all characters are encoded in the same number of bytes. In UTF-8, ASCII characters – i.e. those with code points less than 0x80 (128) – are encoded as they are in ASCII, using a single byte, while code points 0x80 and above are encoded using multiple bytes – up to four per character. This means that not every byte index into a UTF-8 string is necessarily a valid index for a character. If you index into a string at such an invalid byte index, an error is thrown:

In this case, the character is a three-byte character, so the indices 2 and 3 are invalid and the next character's index is 4; this next valid index can be computed by , and the next index after that by and so on.

Because of variable-length encodings, the number of characters in a string (given by) is not always the same as the last index. If you iterate through the indices 1 through and index into , the sequence of characters returned when errors aren't thrown is the sequence of characters comprising the string . Thus we have the identity that , since each character in a string must have its own index. The following is an inefficient and verbose way to iterate through the characters of :

The blank lines actually have spaces on them. Fortunately, the above awkward idiom is unnecessary for iterating through the characters in a string, since you can just use the string as an iterable object, no exception handling required:

Julia uses the UTF-8 encoding by default, and support for new encodings can be added by packages. For example, the LegacyStrings.jl package implements and types. Additional discussion of other encodings and how to implement support for them is beyond the scope of this document for the time being. For further discussion of UTF-8 encoding issues, see the section below on byte array literals. The function is provided to convert data between the various UTF-xx encodings, primarily for working with external data and libraries.

10.4 Concatenation

One of the most common and useful string operations is concatenation:

Julia also provides for string concatenation:

While may seem like a surprising choice to users of languages that provide for string concatenation, this use of has precedent in mathematics, particularly in abstract algebra.

In mathematics, usually denotes a *commutative* operation, where the order of the operands does not matter. An example of this is matrix addition, where for any matrices and that have the same shape. In contrast, typically denotes a *noncommutative* operation, where the order of the operands *does* matter. An example of this is matrix multiplication, where in general . As with matrix multiplication, string concatenation is noncommutative: . As such, is a more natural choice for an infix string concatenation operator, consistent with common mathematical use.

More precisely, the set of all finite-length strings S together with the string concatenation operator forms a free monoid $(S,)$. The identity element of this set is the empty string, . Whenever a free monoid is not commutative, the operation is typically represented as , , or a similar symbol, rather than , which as stated usually implies commutativity.

10.5 Interpolation

Constructing strings using concatenation can become a bit cumbersome, however. To reduce the need for these verbose calls to or repeated multiplications, Julia allows interpolation into string literals using , as in Perl:

This is more readable and convenient and equivalent to the above string concatenation – the system rewrites this apparent single string literal into a concatenation of string literals with variables.

The shortest complete expression after the is taken as the expression whose value is to be interpolated into the string. Thus, you can interpolate any expression into a string using parentheses:

Both concatenation and string interpolation call to convert objects into string form. Most non- objects are converted to strings closely corresponding to how they are entered as literal expressions:

 is the identity for and values, so these are interpolated into strings as themselves, unquoted and unescaped:

To include a literal in a string literal, escape it with a backslash:

10.6 Triple-Quoted String Literals

When strings are created using triple-quotes () they have some special behavior that can be useful for creating longer blocks of text. First, if the opening is followed by a newline, the newline is stripped from the resulting string.

is equivalent to

but

will contain a literal newline at the beginning. Trailing whitespace is left unaltered. They can contain symbols without escaping. Triple-quoted strings are also dedented to the level of the least-indented line. This is useful for defining strings within code that is indented. For example:

In this case the final (empty) line before the closing sets the indentation level.

Note that line breaks in literal strings, whether single- or triple-quoted, result in a newline (LF) character in the string, even if your editor uses a carriage return (CR) or CRLF combination to end lines. To include a CR in a string, use an explicit escape ; for example, you can enter the literal string .

10.7 Common Operations

You can lexicographically compare strings using the standard comparison operators:

You can search for the index of a particular character using the function:

You can start the search for a character at a given offset by providing a third argument:

You can use the function to check if a substring is contained in a string:

The last example shows that can also look for a character literal.

Two other handy string functions are and :

Some other useful functions include:

- gives the maximal (byte) index that can be used to index into .
- the number of characters in .
- gives the first valid index at which a character can be found in (typically 1).
- returns next character at or after the index and the next valid character index following that. With and , can be used to iterate through the characters in .
- gives the number of characters in up to and including any at index .
- gives the index at which the th character in occurs.

10.8 Non-Standard String Literals

There are situations when you want to construct a string or use string semantics, but the behavior of the standard string construct is not quite what is needed. For these kinds of situations, Julia provides non-standard string literals. A non-standard string literal looks like a regular double-quoted string literal, but is immediately prefixed by an identifier, and doesn't behave quite like a normal string literal. Regular expressions, byte array literals and version number literals, as described below, are some examples of non-standard string literals. Other examples are given in the Metaprogramming section.

10.9 Regular Expressions

Julia has Perl-compatible regular expressions (regexes), as provided by the PCRE library. Regular expressions are related to strings in two ways: the obvious connection is that regular expressions are used to find regular patterns in strings; the other connection is that regular expressions are themselves input as strings, which are parsed into a state machine that can be used to efficiently search for patterns in strings. In Julia, regular expressions are input using non-standard string literals prefixed with various identifiers beginning with . The most basic regular expression literal without any options turned on just uses :

To check if a regex matches a string, use :

As one can see here, simply returns true or false, indicating whether the given regex matches the string or not. Commonly, however, one wants to know not just whether a string matched, but also *how* it matched. To capture this information about a match, use the function instead:

If the regular expression does not match the given string, returns – a special value that does not print anything at the interactive prompt. Other than not printing, it is a completely normal value and you can test for it programmatically:

If a regular expression does match, the value returned by is a object. These objects record how the expression matches, including the substring that the pattern matches and any captured substrings, if there are any. This example only captures the portion of the substring that matches, but perhaps we want to capture any non-blank text after the comment character. We could do the following:

When calling , you have the option to specify an index at which to start the search. For example:

You can extract the following info from a object:

- the entire substring matched:
- the captured substrings as an array of strings:
- the offset at which the whole match begins:
- the offsets of the captured substrings as a vector:

For when a capture doesn't match, instead of a substring, contains in that position, and has a zero offset (recall that indices in Julia are 1-based, so a zero offset into a string is invalid). Here is a pair of somewhat contrived examples:

It is convenient to have captures returned as an array so that one can use destructuring syntax to bind them to local variables:

Captures can also be accessed by indexing the object with the number or name of the capture group:

Captures can be referenced in a substitution string when using by using to refer to the nth capture group and prefixing the substitution string with . Capture group 0 refers to the entire match object. Named capture groups can be referenced in the substitution with . For example:

Numbered capture groups can also be referenced as for disambiguation, as in:

You can modify the behavior of regular expressions by some combination of the flags , , , and after the closing double quote mark. These flags have the same meaning as they do in Perl, as explained in this excerpt from the perlre manpage:

For example, the following regex has all three flags turned on:

Triple-quoted regex strings, of the form , are also supported (and may be convenient for regular expressions containing quotation marks or newlines).

10.10 Byte Array Literals

Another useful non-standard string literal is the byte-array string literal: . This form lets you use string notation to express literal byte arrays – i.e. arrays of values. The rules for byte array literals are the following:

- ASCII characters and ASCII escapes produce a single byte.
- and octal escape sequences produce the *byte* corresponding to the escape value.
- Unicode escape sequences produce a sequence of bytes encoding that code point in UTF-8.

There is some overlap between these rules since the behavior of and octal escapes less than 0x80 (128) are covered by both of the first two rules, but here these rules agree. Together, these rules allow one to easily use ASCII characters, arbitrary byte values, and UTF-8 sequences to produce arrays of bytes. Here is an example using all three:

The ASCII string ”DATA” corresponds to the bytes 68, 65, 84, 65. produces the single byte 255. The Unicode escape is encoded in UTF-8 as the three bytes 226, 136, 128. Note that the resulting byte array does not correspond to a valid UTF-8 string – if you try to use this as a regular string literal, you will get a syntax error:

Also observe the significant distinction between and : the former escape sequence encodes the *byte 255*, whereas the latter escape sequence represents the *code point 255*, which is encoded as two bytes in UTF-8:

In character literals, this distinction is glossed over and is allowed to represent the code point 255, because characters *always* represent code points. In strings, however, escapes always represent bytes, not code points, whereas and escapes always represent code points, which are encoded in one or more bytes. For code points less than , it happens that the UTF-8 encoding of each code point is just the single byte produced by the corresponding escape, so the distinction can safely be ignored. For the escapes through as compared to through , however, there is a major difference: the former escapes all encode single bytes, which – unless followed by very specific continuation bytes – do not form valid UTF-8 data, whereas the latter escapes all represent Unicode code points with two-byte encodings.

If this is all extremely confusing, try reading ”The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets”. It’s an excellent introduction to Unicode and UTF-8, and may help alleviate some confusion regarding the matter.

10.11 Version Number Literals

Version numbers can easily be expressed with non-standard string literals of the form . Version number literals create objects which follow the specifications of semantic versioning, and therefore are composed of major, minor and patch numeric values, followed by pre-release and build alpha-numeric annotations. For example, is broken into major version , minor version , patch version , pre-release and build . When entering a version literal, everything except the major version number is optional, therefore e.g. is equivalent to (with empty pre-release/build annotations), is equivalent to , and so on.

objects are mostly useful to easily and correctly compare two (or more) versions. For example, the constant holds Julia version number as a object, and therefore one can define some version-specific behavior using simple statements as:

Note that in the above example the non-standard version number is used, with a trailing : this notation is a Julia extension of the standard, and it's used to indicate a version which is lower than any release, including all of its pre-releases. So in the above example the code would only run with stable versions, and exclude such versions as . In order to also allow for unstable (i.e. pre-release) versions, the lower bound check should be modified like this: .

Another non-standard version specification extension allows one to use a trailing to express an upper limit on build versions, e.g. can be used to mean any version above and any of its builds: it will return for version and for .

It is good practice to use such special versions in comparisons (particularly, the trailing should always be used on upper bounds unless there's a good reason not to), but they must not be used as the actual version number of anything, as they are invalid in the semantic versioning scheme.

Besides being used for the constant, objects are widely used in the module, to specify packages versions and their dependencies.

10.12 Raw String Literals

Raw strings without interpolation or unescaping can be expressed with non-standard string literals of the form . Raw string literals create ordinary objects which contain the enclosed contents exactly as entered with no interpolation or unescaping. This is useful for strings which contain code or markup in other languages which use or as special characters. The exception is quotation marks that still must be escaped, e.g. is equivalent to .

Chapter 11

Functions

In Julia, a function is an object that maps a tuple of argument values to a return value. Julia functions are not pure mathematical functions, in the sense that functions can alter and be affected by the global state of the program. The basic syntax for defining functions in Julia is:

There is a second, more terse syntax for defining a function in Julia. The traditional function declaration syntax demonstrated above is equivalent to the following compact "assignment form":

In the assignment form, the body of the function must be a single expression, although it can be a compound expression (see Compound Expressions). Short, simple function definitions are common in Julia. The short function syntax is accordingly quite idiomatic, considerably reducing both typing and visual noise.

A function is called using the traditional parenthesis syntax:

Without parentheses, the expression refers to the function object, and can be passed around like any value:

As with variables, Unicode can also be used for function names:

11.1 Argument Passing Behavior

Julia function arguments follow a convention sometimes called ”pass-by-sharing”, which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable *bindings* (new locations that can refer to values), but the values they refer to are identical to the passed values. Modifications to mutable values (such as s) made within a function will be visible to the caller. This is the same behavior found in Scheme, most Lisps, Python, Ruby and Perl, among other dynamic languages.

11.2 The Keyword

The value returned by a function is the value of the last expression evaluated, which, by default, is the last expression in the body of the function definition. In the example function, , from the previous section this is the value of the expression . As in C and most other imperative or functional languages, the keyword causes a function to return immediately, providing an expression whose value is returned:

Since function definitions can be entered into interactive sessions, it is easy to compare these definitions:

Of course, in a purely linear function body like , the usage of is pointless since the expression is never evaluated and we could simply make the last expression in the function and omit the . In conjunction with other control flow, however, is of real use. Here, for example, is a function that computes the hypotenuse length of a right triangle with sides of length and , avoiding overflow:

There are three possible points of return from this function, returning the values of three different expressions, depending on the values of and . The on the last line could be omitted since it is the last expression.

11.3 Operators Are Functions

In Julia, most operators are just functions with support for special syntax. (The exceptions are operators with special evaluation semantics like and . These operators cannot be functions since Short-Circuit Evaluation requires that their operands are not evaluated before evaluation of the operator.) Accordingly, you can also apply them using parenthesized argument lists, just as you would any other function:

The infix form is exactly equivalent to the function application form – in fact the former is parsed to produce the function call internally. This also means that you can assign and pass around operators such as and just like you would with other function values:

Under the name , the function does not support infix notation, however.

11.4 Operators With Special Names

A few special expressions correspond to calls to functions with non-obvious names. These are:

Expression	Calls

11.5 Anonymous Functions

Functions in Julia are first-class objects: they can be assigned to variables, and called using the standard function call syntax from the variable they have been assigned to. They can be used as arguments, and they can be returned as values. They can also be created anonymously, without being given a name, using either of these syntaxes:

This creates a function taking one argument and returning the value of the polynomial at that value. Notice that the result is a generic function, but with a compiler-generated name based on consecutive numbering.

The primary use for anonymous functions is passing them to functions which take other functions as arguments. A classic example is , which applies a function to each value of an array and returns a new array containing the resulting values:

This is fine if a named function effecting the transform already exists to pass as the first argument to . Often, however, a ready-to-use, named function does not exist. In these situations, the anonymous function construct allows easy creation of a single-use function object without needing a name:

An anonymous function accepting multiple arguments can be written using the syntax . A zero-argument anonymous function is written as . The idea of a function with no arguments may seem strange, but is useful for ”delaying” a computation. In this usage, a block of code is wrapped in a zero-argument function, which is later invoked by calling it as .

11.6 Multiple Return Values

In Julia, one returns a tuple of values to simulate returning multiple values. However, tuples can be created and destructured without needing parentheses, thereby providing an illusion that multiple values are being returned, rather than a single tuple value. For example, the following function returns a pair of values:

If you call it in an interactive session without assigning the return value anywhere, you will see the tuple returned:

A typical usage of such a pair of return values, however, extracts each value into a variable. Julia supports simple tuple ”destructuring” that facilitates this:

You can also return multiple values via an explicit usage of the keyword:

This has the exact same effect as the previous definition of .

11.7 Varargs Functions

It is often convenient to be able to write functions taking an arbitrary number of arguments. Such functions are traditionally known as ”varargs” functions, which is short for ”variable number of arguments”. You can define a varargs function by following the last argument with an ellipsis:

The variables and are bound to the first two argument values as usual, and the variable is bound to an iterable collection of the zero or more values passed to after its first two arguments:

In all these cases, is bound to a tuple of the trailing values passed to .

It is possible to constrain the number of values passed as a variable argument; this will be discussed later in Parametrically-constrained Varargs methods.

On the flip side, it is often handy to ”splice” the values contained in an iterable collection into a function call as individual arguments. To do this, one also uses but in the function call instead:

In this case a tuple of values is spliced into a varargs call precisely where the variable number of arguments go. This need not be the case, however:

Furthermore, the iterable object spliced into a function call need not be a tuple:

Also, the function that arguments are spliced into need not be a varargs function (although it often is):

As you can see, if the wrong number of elements are in the spliced container, then the function call will fail, just as it would if too many arguments were given explicitly.

11.8 Optional Arguments

In many cases, function arguments have sensible default values and therefore might not need to be passed explicitly in every call. For example, the library function interprets a string as a number in some base. The argument defaults to . This behavior can be expressed concisely as:

With this definition, the function can be called with either two or three arguments, and is automatically passed when a third argument is not specified:

Optional arguments are actually just a convenient syntax for writing multiple method definitions with different numbers of arguments (see Note on Optional and keyword Arguments).

11.9 Keyword Arguments

Some functions need a large number of arguments, or have a large number of behaviors. Remembering how to call such functions can be difficult. Keyword arguments can make these complex interfaces easier to use and extend by allowing arguments to be identified by name instead of only by position.

For example, consider a function that plots a line. This function might have many options, for controlling line style, width, color, and so on. If it accepts keyword arguments, a possible call might look like , where we have chosen to specify only line width. Notice that this serves two purposes. The call is easier to read, since we can label an argument with its meaning. It also becomes possible to pass any subset of a large number of arguments, in any order.

Functions with keyword arguments are defined using a semicolon in the signature:

When the function is called, the semicolon is optional: one can either call or , but the former style is more common. An explicit semicolon is required only for passing varargs or computed keywords as described below.

Keyword argument default values are evaluated only when necessary (when a corresponding keyword argument is not passed), and in left-to-right order. Therefore default expressions may refer to prior keyword arguments.

The types of keyword arguments can be made explicit as follows:

Extra keyword arguments can be collected using , as in varargs functions:

Inside , will be a collection of tuples, where each is a symbol. Such collections can be passed as keyword arguments using a semicolon in a call, e.g. . Dictionaries can also be used for this purpose.

One can also pass tuples, or any iterable expression (such as a pair) that can be assigned to such a tuple, explicitly after a semicolon. For example, and are equivalent to . This is useful in situations where the keyword name is computed at runtime.

The nature of keyword arguments makes it possible to specify the same argument more than once. For example, in the call it is possible that the structure also contains a value for . In such a case the rightmost occurrence takes precedence; in this example, is certain to have the value .

11.10 Evaluation Scope of Default Values

When optional and keyword argument default expressions are evaluated, only *previous* arguments are in scope. For example, given this definition:

the in refers to a in an outer scope, not the subsequent argument .

11.11 Do-Block Syntax for Function Arguments

Passing functions as arguments to other functions is a powerful technique, but the syntax for it is not always convenient. Such calls are especially awkward to write when the function argument requires multiple lines. As an example, consider calling on a function with several cases:

Julia provides a reserved word for rewriting this code more clearly:

The syntax creates an anonymous function with argument and passes it as the first argument to . Similarly, would create a two-argument anonymous function, and a plain would declare that what follows is an anonymous function of the form .

How these arguments are initialized depends on the ”outer” function; here, will sequentially set to , , , calling the anonymous function on each, just as would happen in the syntax .

This syntax makes it easier to use functions to effectively extend the language, since calls look like normal code blocks. There are many possible uses quite different from , such as managing system state. For example, there is a version of that runs code ensuring that the opened file is eventually closed:

This is accomplished by the following definition:

Here, first opens the file for writing and then passes the resulting output stream to the anonymous function you defined in the block. After your function exits, will make sure that the stream is properly closed, regardless of whether your function exited normally or threw an exception. (The construct will be described in Control Flow.)

With the block syntax, it helps to check the documentation or implementation to know how the arguments of the user function are initialized.

11.12 Dot Syntax for Vectorizing Functions

In technical-computing languages, it is common to have ”vectorized” versions of functions, which simply apply a given function to each element of an array to yield a new array via . This kind of syntax is convenient for data processing, but in other languages vectorization is also often required for performance: if loops are slow, the ”vectorized” version of a function can call fast library code written in a low-level language. In Julia, vectorized functions are *not* required for performance, and indeed it is often beneficial to write your own loops (see Performance Tips), but they can still be convenient. Therefore, *any* Julia function can be applied elementwise to any array (or other collection) with the syntax . For example can be applied to all elements in the vector , like so:

Of course, you can omit the dot if you write a specialized ”vector” method of , e.g. via , and this is just as efficient as . But that approach requires you to decide in advance which functions you want to vectorize.

More generally, is actually equivalent to , which allows you to operate on multiple arrays (even of different shapes), or a mix of arrays and scalars (see Broadcasting). For example, if you have , then will return a new array consisting of for each in , and will return a new vector consisting of for each index (throwing an exception if the vectors have different length).

Moreover, *nested* calls are *fused* into a single loop. For example, is equivalent to , similar to : there is only a single loop over , and a single array is allocated for the result. [In contrast, in a typical "vectorized" language would first allocate one temporary array for , and then compute in a separate loop, allocating a second array.] This loop fusion is not a compiler optimization that may or may not occur, it is a *syntactic guarantee* whenever nested calls are encountered. Technically, the fusion stops as soon as a "non-dot" function call is encountered; for example, in the and loops cannot be merged because of the intervening function.

Finally, the maximum efficiency is typically achieved when the output array of a vectorized operation is *pre-allocated*, so that repeated calls do not allocate new arrays over and over again for the results (see Pre-allocating outputs). A convenient syntax for this is , which is equivalent to except that, as above, the loop is fused with any nested "dot" calls. For example, is equivalent to , overwriting with in-place. If the left-hand side is an array-indexing expression, e.g. , then it translates to on a , e.g. , so that the left-hand side is updated in-place.

Since adding dots to many operations and function calls in an expression can be tedious and lead to code that is difficult to read, the macro is provided to convert *every* function call, operation, and assignment in an expression into the "dotted" version.

Binary (or unary) operators like are handled with the same mechanism: they are equivalent to calls and are fused with other nested "dot" calls. etcetera is equivalent to and results in a fused in-place assignment; see also dot operators.

11.13 Further Reading

We should mention here that this is far from a complete picture of defining functions. Julia has a sophisticated type system and allows multiple dispatch on argument types. None of the examples given here provide any type annotations on their arguments, meaning that they are applicable to all types of arguments. The type system is described in Types and defining a function in terms of methods chosen by multiple dispatch on run-time argument types is described in Methods.

Chapter 12

Control Flow

Julia provides a variety of control flow constructs:

- Compound Expressions: and .
- Conditional Evaluation: -- and (ternary operator).
- Short-Circuit Evaluation: , and chained comparisons.
- Repeated Evaluation: Loops: and .
- Exception Handling: -, and .
- Tasks (aka Coroutines): .

The first five control flow mechanisms are standard to high-level programming languages. s are not so standard: they provide non-local control flow, making it possible to switch between temporarily-suspended computations. This is a powerful construct: both exception handling and cooperative multitasking are implemented in Julia using tasks. Everyday programming requires no direct usage of tasks, but certain problems can be solved much more easily by using tasks.

12.1 Compound Expressions

Sometimes it is convenient to have a single expression which evaluates several subexpressions in order, returning the value of the last subexpression as its value. There are two Julia constructs that accomplish this: blocks and chains. The value of both compound expression constructs is that of the last subexpression. Here's an example of a block:

Since these are fairly small, simple expressions, they could easily be placed onto a single line, which is where the chain syntax comes in handy:

This syntax is particularly useful with the terse single-line function definition form introduced in Functions. Although it is typical, there is no requirement that blocks be multiline or that chains be single-line:

12.2 Conditional Evaluation

Conditional evaluation allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the -- conditional syntax:

If the condition expression is , then the corresponding block is evaluated; otherwise the condition expression is evaluated, and if it is , the corresponding block is evaluated; if neither expression is true, the block is evaluated. Here it is in action:

The and blocks are optional, and as many blocks as desired can be used. The condition expressions in the -- construct are evaluated until the first one evaluates to , after which the associated block is evaluated, and no further condition expressions or blocks are evaluated.

 blocks are ”leaky”, i.e. they do not introduce a local scope. This means that new variables defined inside the clauses can be used after the block, even if they weren’t defined before. So, we could have defined the function above as

The variable is declared inside the block, but used outside. However, when depending on this behavior, make sure all possible code paths define a value for the variable. The following change to the above function results in a runtime error

 blocks also return a value, which may seem unintuitive to users coming from many other languages. This value is simply the return value of the last executed statement in the branch that was chosen, so

Note that very short conditional statements (one-liners) are frequently expressed using Short-Circuit Evaluation in Julia, as outlined in the next section.

Unlike C, MATLAB, Perl, Python, and Ruby – but like Java, and a few other stricter, typed languages – it is an error if the value of a conditional expression is anything but or :

This error indicates that the conditional was of the wrong type: rather than the required .

The so-called ”ternary operator”, , is closely related to the -- syntax, but is used where a conditional choice between single expression values is required, as opposed to conditional execution of longer blocks of code. It gets its name from being the only operator in most languages taking three operands:

The expression , before the , is a condition expression, and the ternary operation evaluates the expression , before the , if the condition is or the expression , after the , if it is . Note that the spaces around and are mandatory: an expression like is not a valid ternary expression (but a newline is acceptable after both the and the).

The easiest way to understand this behavior is to see an example. In the previous example, the call is shared by all three branches: the only real choice is which literal string to print. This could be written more concisely using the ternary operator. For the sake of clarity, let’s try a two-way version first:

If the expression is true, the entire ternary operator expression evaluates to the string and otherwise it evaluates to the string . The original three-way example requires chaining multiple uses of the ternary operator together:

To facilitate chaining, the operator associates from right to left.

It is significant that like --, the expressions before and after the are only evaluated if the condition expression evaluates to or , respectively:

12.3 Short-Circuit Evaluation

Short-circuit evaluation is quite similar to conditional evaluation. The behavior is found in most imperative programming languages having the and boolean operators: in a series of boolean expressions connected by these operators, only the minimum number of expressions are evaluated as are necessary to determine the final boolean value of the entire chain. Explicitly, this means that:

- In the expression , the subexpression is only evaluated if evaluates to .
- In the expression , the subexpression is only evaluated if evaluates to .

The reasoning is that must be if is , regardless of the value of , and likewise, the value of must be true if is , regardless of the value of . Both and associate to the right, but has higher precedence than does. It's easy to experiment with this behavior:

You can easily experiment in the same way with the associativity and precedence of various combinations of and operators.

This behavior is frequently used in Julia to form an alternative to very short statements. Instead of , one can write (which could be read as: <cond> *and then* <statement>). Similarly, instead of , one can write (which could be read as: <cond> *or else* <statement>).

For example, a recursive factorial routine could be defined like this:

Boolean operations *without* short-circuit evaluation can be done with the bitwise boolean operators introduced in Mathematical Operations and Elementary Functions: and . These are normal functions, which happen to support infix operator syntax, but always evaluate their arguments:

Just like condition expressions used in , or the ternary operator, the operands of or must be boolean values (or). Using a non-boolean value anywhere except for the last entry in a conditional chain is an error:

On the other hand, any type of expression can be used at the end of a conditional chain. It will be evaluated and returned depending on the preceding conditionals:

12.4 Repeated Evaluation: Loops

There are two constructs for repeated evaluation of expressions: the loop and the loop. Here is an example of a loop:

The loop evaluates the condition expression (in this case), and as long it remains , keeps also evaluating the body of the loop. If the condition expression is when the loop is first reached, the body is never evaluated.

The loop makes common repeated evaluation idioms easier to write. Since counting up and down like the above loop does is so common, it can be expressed more concisely with a loop:

Here the is a object, representing the sequence of numbers 1, 2, 3, 4, 5. The loop iterates through these values, assigning each one in turn to the variable . One rather important distinction between the previous loop form and the loop form is the scope during which the variable is visible. If the variable has not been introduced in an other scope, in the loop form, it is visible only inside of the loop, and not afterwards. You'll either need a new interactive session instance or a different variable name to test this:

See Scope of Variables for a detailed explanation of variable scope and how it works in Julia.

In general, the loop construct can iterate over any container. In these cases, the alternative (but fully equivalent) keyword or is typically used instead of , since it makes the code read more clearly:

Various types of iterable containers will be introduced and discussed in later sections of the manual (see, e.g., Multi-dimensional Arrays).

It is sometimes convenient to terminate the repetition of a before the test condition is falsified or stop iterating in a loop before the end of the iterable object is reached. This can be accomplished with the keyword:

Without the keyword, the above loop would never terminate on its own, and the loop would iterate up to 1000. These loops are both exited early by using .

In other circumstances, it is handy to be able to stop an iteration and move on to the next one immediately. The keyword accomplishes this:

This is a somewhat contrived example since we could produce the same behavior more clearly by negating the condition and placing the call inside the block. In realistic usage there is more code to be evaluated after the , and often there are multiple points from which one calls .

Multiple nested loops can be combined into a single outer loop, forming the cartesian product of its iterables:

A statement inside such a loop exits the entire nest of loops, not just the inner one.

12.5 Exception Handling

When an unexpected condition occurs, a function may be unable to return a reasonable value to its caller. In such cases, it may be best for the exceptional condition to either terminate the program, printing a diagnostic error message, or if the programmer has provided code to handle such exceptional circumstances, allow that code to take the appropriate action.

Built-in s

s are thrown when an unexpected condition has occurred. The built-in s listed below all interrupt the normal flow of control.

For example, the function throws a if applied to a negative real value:

You may define your own exceptions in the following way:

The function

Exceptions can be created explicitly with . For example, a function defined only for nonnegative numbers could be written to a if the argument is negative:

Note that without parentheses is not an exception, but a type of exception. It needs to be called to obtain an object:

Additionally, some exception types take one or more arguments that are used for error reporting:

This mechanism can be implemented easily by custom exception types following the way is written:

Note

When writing an error message, it is preferred to make the first word lowercase. For example,

is preferred over

.

However, sometimes it makes sense to keep the uppercase first letter, for instance if an argument to a function is a capital letter: .

Errors

The function is used to produce an that interrupts the normal flow of control.

Suppose we want to stop execution immediately if the square root of a negative number is taken. To do this, we can define a fussy version of the function that raises an error if its argument is negative:

If is called with a negative value from another function, instead of trying to continue execution of the calling function, it returns immediately, displaying the error message in the interactive session:

Warnings and informational messages

Julia also provides other functions that write messages to the standard error I/O, but do not throw any s and hence do not interrupt execution:

The statement

The statement allows for s to be tested for. For example, a customized square root function can be written to automatically call either the real or complex square root method on demand using s :

It is important to note that in real code computing this function, one would compare to zero instead of catching an exception. The exception is much slower than simply comparing and branching.

 statements also allow the to be saved in a variable. In this contrived example, the following example calculates the square root of the second element of if is indexable, otherwise assumes is a real number and returns its square root:

Note that the symbol following will always be interpreted as a name for the exception, so care is needed when writing expressions on a single line. The following code will *not* work to return the value of in case of an error:

Instead, use a semicolon or insert a line break after :

The clause is not strictly necessary; when omitted, the default return value is .

The power of the construct lies in the ability to unwind a deeply nested computation immediately to a much higher level in the stack of calling functions. There are situations where no error has occurred, but the ability to unwind the stack and pass a value to a higher level is desirable. Julia provides the , and functions for more advanced error handling.

Clauses

In code that performs state changes or uses resources like files, there is typically clean-up work (such as closing files) that needs to be done when the code is finished. Exceptions potentially complicate this task, since they can cause a block of code to exit before reaching its normal end. The keyword provides a way to run some code when a given block of code exits, regardless of how it exits.

For example, here is how we can guarantee that an opened file is closed:

When control leaves the block (for example due to a , or just finishing normally), will be executed. If the block exits due to an exception, the exception will continue propagating. A block may be combined with and as well. In this case the block will run after has handled the error.

12.6 Tasks (aka Coroutines)

Tasks are a control flow feature that allows computations to be suspended and resumed in a flexible manner. This feature is sometimes called by other names, such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations.

When a piece of computing work (in practice, executing a particular function) is designated as a , it becomes possible to interrupt it by switching to another . The original can later be resumed, at which point it will pick up right where it left off. At first, this may seem similar to a function call. However there are two key differences. First, switching tasks does not use any space, so any number of task switches can occur without consuming the call stack. Second, switching among tasks can occur in any order, unlike function calls, where the called function must finish executing before control returns to the calling function.

This kind of control flow can make it much easier to solve certain problems. In some problems, the various pieces of required work are not naturally related by function calls; there is no obvious ”caller” or ”callee” among the jobs that need to be done. An example is the producer-consumer problem, where one complex procedure is generating values and another complex procedure is consuming them. The consumer cannot simply call a producer function to get a value, because the producer may have more values to generate and so might not yet be ready to return. With tasks, the producer and consumer can both run as long as they need to, passing values back and forth as necessary.

Julia provides a mechanism for solving this problem. A is a waitable first-in first-out queue which can have multiple tasks reading from and writing to it.

Let’s define a producer task, which produces values via the call. To consume values, we need to schedule the producer to run in a new task. A special constructor which accepts a 1-arg function as an argument can be used to run a task bound to a channel. We can then values repeatedly from the channel object:

One way to think of this behavior is that was able to return multiple times. Between calls to , the producer's execution is suspended and the consumer has control.

The returned can be used as an iterable object in a loop, in which case the loop variable takes on all the produced values. The loop is terminated when the channel is closed.

Note that we did not have to explicitly close the channel in the producer. This is because the act of binding a to a associates the open lifetime of a channel with that of the bound task. The channel object is closed automatically when the task terminates. Multiple channels can be bound to a task, and vice-versa.

While the constructor expects a 0-argument function, the method which creates a channel bound task expects a function that accepts a single argument of type . A common pattern is for the producer to be parameterized, in which case a partial function application is needed to create a 0 or 1 argument anonymous function.

For objects this can be done either directly or by use of a convenience macro:

To orchestrate more advanced work distribution patterns, and can be used in conjunction with and constructors to explicitly link a set of channels with a set of producer/consumer tasks.

Note that currently Julia tasks are not scheduled to run on separate CPU cores. True kernel threads are discussed under the topic of Parallel Computing.

Core task operations

Let us explore the low level construct to underestand how task switching works. suspends the current task, switches to the specified , and causes that task's last call to return the specified . Notice that is the only operation required to use task-style control flow; instead of calling and returning we are always just switching to a different task. This is why this feature is also called "symmetric coroutines"; each task is switched to and from using the same mechanism.

is powerful, but most uses of tasks do not invoke it directly. Consider why this might be. If you switch away from the current task, you will probably want to switch back to it at some point, but knowing when to switch back, and knowing which task has the responsibility of switching back, can require considerable coordination. For example, and are blocking operations, which, when used in the context of channels maintain state to remember who the consumers are. Not needing to manually keep track of the consuming task is what makes easier to use than the low-level .

In addition to , a few other basic functions are needed to use tasks effectively.

- gets a reference to the currently-running task.
- queries whether a task has exited.
- queries whether a task has run yet.
- manipulates a key-value store specific to the current task.

Tasks and events

Most task switches occur as a result of waiting for events such as I/O requests, and are performed by a scheduler included in the standard library. The scheduler maintains a queue of runnable tasks, and executes an event loop that restarts tasks based on external events such as message arrival.

The basic function for waiting for an event is . Several objects implement ; for example, given a object, will wait for it to exit. is often implicit; for example, a can happen inside a call to to wait for data to be available.

In all of these cases, ultimately operates on a object, which is in charge of queueing and restarting tasks. When a task calls on a , the task is marked as non-runnable, added to the condition's queue, and switches to the scheduler. The scheduler will then pick another task to run, or block waiting for external events. If all goes well, eventually an event handler will call on the condition, which causes tasks waiting for that condition to become runnable again.

A task created explicitly by calling is initially not known to the scheduler. This allows you to manage tasks manually using if you wish. However, when such a task waits for an event, it still gets restarted automatically when the event happens, as you would expect. It is also possible to make the scheduler run a task whenever it can, without necessarily waiting for any events. This is done by calling , or using the or macros (see Parallel Computing for more details).

Task states

Tasks have a field that describes their execution status. A is one of the following symbols:

Symbol	Meaning
	Currently running, or available to be switched to
	Blocked waiting for a specific event
	In the scheduler's run queue about to be restarted
	Successfully finished executing
	Finished with an uncaught exception

Chapter 13

Scope of Variables

The *scope* of a variable is the region of code within which a variable is visible. Variable scoping helps avoid variable naming conflicts. The concept is intuitive: two functions can both have arguments called without the two 's referring to the same thing. Similarly there are many other cases where different blocks of code can use the same name without referring to the same thing. The rules for when the same variable name does or doesn't refer to the same thing are called scope rules; this section spells them out in detail.

Certain constructs in the language introduce *scope blocks*, which are regions of code that are eligible to be the scope of some set of variables. The scope of a variable cannot be an arbitrary set of source lines; instead, it will always line up with one of these blocks. There are two main types of scopes in Julia, *global scope* and *local scope*, the latter can be nested. The constructs introducing scope blocks are:

Scope name	block/construct introducing this kind of scope
Global Scope	, , at interactive prompt (REPL)
Local Scope	Soft Local Scope: , , comprehensions, try-catch-finally,
Local Scope	Hard Local Scope: functions (either syntax, anonymous & do-blocks), ,

Notably missing from this table are begin blocks and if blocks, which do *not* introduce new scope blocks. All three types of scopes follow somewhat different rules which will be explained below as well as some extra rules for certain blocks.

Julia uses lexical scoping, meaning that a function's scope does not inherit from its caller's scope, but from the scope in which the function was defined. For example, in the following code the inside refers to the in the global scope of its module :

```

```

and not a in the scope where is used:

```

```

Thus *lexical scope* means that the scope of variables can be inferred from the source code alone.

13.1 Global Scope

Each module introduces a new global scope, separate from the global scope of all other modules; there is no all-encompassing global scope. Modules can introduce variables of other modules into their scope through the using or import statements or through qualified access using the dot-notation, i.e. each module is a so-called *namespace*. Note that variable bindings can only be changed within their global scope and not from an outside module.

Note that the interactive prompt (aka REPL) is in the global scope of the module .

13.2 Local Scope

A new local scope is introduced by most code-blocks, see above table for a complete list. A local scope *usually* inherits all the variables from its parent scope, both for reading and writing. There are two subtypes of local scopes, hard and soft, with slightly different rules concerning what variables are inherited. Unlike global scopes, local scopes are not namespaces, thus variables in an inner scope cannot be retrieved from the parent scope through some sort of qualified access.

The following rules and examples pertain to both hard and soft local scopes. A newly introduced variable in a local scope does not back-propagate to its parent scope. For example, here the is not introduced into the top-level scope:

(Note, in this and all following examples it is assumed that their top-level is a global scope with a clean workspace, for instance a newly started REPL.)

Inside a local scope a variable can be forced to be a local variable using the keyword:

Inside a local scope a new global variable can be defined using the keyword :

The location of both the and keywords within the scope block is irrelevant. The following is equivalent to the last example (although stylistically worse):

The and keywords can also be applied to destructuring assignments, e.g. . In this case the keyword affects all listed variables.

Soft Local Scope

> In a soft local scope, all variables are inherited from its parent scope unless a variable is specifically marked with the keyword .

Soft local scopes are introduced by for-loops, while-loops, comprehensions, try-catch-finally-blocks, and let-blocks. There are some extra rules for Let Blocks and for For Loops and Comprehensions.

In the following example the and refer always to the same variables as the soft local scope inherits both read and write variables:

Within soft scopes, the *global* keyword is never necessary, although allowed. The only case when it would change the semantics is (currently) a syntax error:

Hard Local Scope

Hard local scopes are introduced by function definitions (in all their forms), struct type definition blocks, and macro-definitions.

In a hard local scope, all variables are inherited from its parent scope unless:

- an assignment would result in a modified *global* variable, or
- a variable is specifically marked with the keyword .

Thus global variables are only inherited for reading but not for writing:

An explicit is needed to assign to a global variable:

Note that *nested functions* can behave differently to functions defined in the global scope as they can modify their parent scope's *local* variables:

The distinction between inheriting global and local variables for assignment can lead to some slight differences between functions defined in local vs. global scopes. Consider the modification of the last example by moving to the global scope:

Note that above subtlety does not pertain to type and macro definitions as they can only appear at the global scope. There are special scoping rules concerning the evaluation of default and keyword function arguments which are described in the Function section.

An assignment introducing a variable used inside a function, type or macro definition need not come before its inner usage:

This behavior may seem slightly odd for a normal variable, but allows for named functions – which are just normal variables holding function objects – to be used before they are defined. This allows functions to be defined in whatever order is intuitive and convenient, rather than forcing bottom up ordering or requiring forward declarations, as long as they are defined by the time they are actually called. As an example, here is an inefficient, mutually recursive way to test if positive integers are even or odd:

Julia provides built-in, efficient functions to test for oddness and evenness called and so the above definitions should only be taken as examples.

Hard vs. Soft Local Scope

Blocks which introduce a soft local scope, such as loops, are generally used to manipulate the variables in their parent scope. Thus their default is to fully access all variables in their parent scope.

Conversely, the code inside blocks which introduce a hard local scope (function, type, and macro definitions) can be executed at any place in a program. Remotely changing the state of global variables in other modules should be done with care and thus this is an opt-in feature requiring the keyword.

The reason to allow *modifying local* variables of parent scopes in nested functions is to allow constructing closures which have a private state, for instance the variable in the following example:

See also the closures in the examples in the next two sections.

Let Blocks

Unlike assignments to local variables, statements allocate new variable bindings each time they run. An assignment modifies an existing value location, and creates new locations. This difference is usually not important, and is only detectable in the case of variables that outlive their scope via closures. The syntax accepts a comma-separated series of assignments and variable names:

The assignments are evaluated in order, with each right-hand side evaluated in the scope before the new variable on the left-hand side has been introduced. Therefore it makes sense to write something like since the two variables are distinct and have separate storage. Here is an example where the behavior of is needed:

Here we create and store two closures that return variable . However, it is always the same variable , so the two closures behave identically. We can use to create a new binding for :

Since the construct does not introduce a new scope, it can be useful to use a zero-argument to just introduce a new scope block without creating any new bindings:

Since introduces a new scope block, the inner local is a different variable than the outer local .

For Loops and Comprehensions

 loops and Comprehensions have the following behavior: any new variables introduced in their body scopes are freshly allocated for each loop iteration. This is in contrast to loops which reuse the variables for all iterations. Therefore these constructs are similar to loops with blocks inside:

 loops will reuse existing variables for its iteration variable:

However, comprehensions do not do this, and always freshly allocate their iteration variables:

13.3 Constants

A common use of variables is giving names to specific, unchanging values. Such variables are only assigned once. This intent can be conveyed to the compiler using the keyword:

The declaration is allowed on both global and local variables, but is especially useful for globals. It is difficult for the compiler to optimize code involving global variables, since their values (or even their types) might change at almost any time. If a global variable will not change, adding a declaration solves this performance problem.

Local constants are quite different. The compiler is able to determine automatically when a local variable is constant, so local constant declarations are not necessary for performance purposes.

Special top-level assignments, such as those performed by the and keywords, are constant by default.

Note that only affects the variable binding; the variable may be bound to a mutable object (such as an array), and that object may still be modified.

Chapter 14

Types

Type systems have traditionally fallen into two quite different camps: static type systems, where every program expression must have a type computable before the execution of the program, and dynamic type systems, where nothing is known about types until run time, when the actual values manipulated by the program are available. Object orientation allows some flexibility in statically typed languages by letting code be written without the precise types of values being known at compile time. The ability to write code that can operate on different types is called polymorphism. All code in classic dynamically typed languages is polymorphic: only by explicitly checking types, or when objects fail to support operations at run-time, are the types of any values ever restricted.

Julia's type system is dynamic, but gains some of the advantages of static type systems by making it possible to indicate that certain values are of specific types. This can be of great assistance in generating efficient code, but even more significantly, it allows method dispatch on the types of function arguments to be deeply integrated with the language. Method dispatch is explored in detail in Methods, but is rooted in the type system presented here.

The default behavior in Julia when types are omitted is to allow values to be of any type. Thus, one can write many useful Julia programs without ever explicitly using types. When additional expressiveness is needed, however, it is easy to gradually introduce explicit type annotations into previously "untyped" code. Doing so will typically increase both the performance and robustness of these systems, and perhaps somewhat counterintuitively, often significantly simplify them.

Describing Julia in the lingo of type systems, it is: dynamic, nominative and parametric. Generic types can be parameterized, and the hierarchical relationships between types are explicitly declared, rather than implied by compatible structure. One particularly distinctive feature of Julia's type system is that concrete types may not subtype each other: all concrete types are final and may only have abstract types as their supertypes. While this might at first seem unduly restrictive, it has many beneficial consequences with surprisingly few drawbacks. It turns out that being able to inherit behavior is much more important than being able to inherit structure, and inheriting both causes significant difficulties in traditional object-oriented languages. Other high-level aspects of Julia's type system that should be mentioned up front are:

- There is no division between object and non-object values: all values in Julia are true objects having a type that belongs to a single, fully connected type graph, all nodes of which are equally first-class as types.
- There is no meaningful concept of a "compile-time type": the only type a value has is its actual type when the program is running. This is called a "run-time type" in object-oriented languages where the combination of static compilation with polymorphism makes this distinction significant.
- Only values, not variables, have types – variables are simply names bound to values.
- Both abstract and concrete types can be parameterized by other types. They can also be parameterized by symbols, by values of any type for which returns true (essentially, things like numbers and bools that are stored like

C types or structs with no pointers to other objects), and also by tuples thereof. Type parameters may be omitted when they do not need to be referenced or restricted.

Julia's type system is designed to be powerful and expressive, yet clear, intuitive and unobtrusive. Many Julia programmers may never feel the need to write code that explicitly uses types. Some kinds of programming, however, become clearer, simpler, faster and more robust with declared types.

14.1 Type Declarations

The operator can be used to attach type annotations to expressions and variables in programs. There are two primary reasons to do this:

1. As an assertion to help confirm that your program works the way you expect,
2. To provide extra type information to the compiler, which can then improve performance in some cases

When appended to an expression computing a value, the operator is read as "is an instance of". It can be used anywhere to assert that the value of the expression on the left is an instance of the type on the right. When the type on the right is concrete, the value on the left must have that type as its implementation – recall that all concrete types are final, so no implementation is a subtype of any other. When the type is abstract, it suffices for the value to be implemented by a concrete type that is a subtype of the abstract type. If the type assertion is not true, an exception is thrown, otherwise, the left-hand value is returned:

This allows a type assertion to be attached to any expression in-place.

When appended to a variable on the left-hand side of an assignment, or as part of a declaration, the operator means something a bit different: it declares the variable to always have the specified type, like a type declaration in a statically-typed language such as C. Every value assigned to the variable will be converted to the declared type using :

This feature is useful for avoiding performance "gotchas" that could occur if one of the assignments to a variable changed its type unexpectedly.

This "declaration" behavior only occurs in specific contexts:

and applies to the whole current scope, even before the declaration. Currently, type declarations cannot be used in global scope, e.g. in the REPL, since Julia does not yet have constant-type globals.

Declarations can also be attached to function definitions:

Returning from this function behaves just like an assignment to a variable with a declared type: the value is always converted to .

14.2 Abstract Types

Abstract types cannot be instantiated, and serve only as nodes in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. We begin with abstract types even though they have no instantiation because they are the backbone of the type system: they form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations.

Recall that in Integers and Floating-Point Numbers, we introduced a variety of concrete types of numeric values: , , , , , , , , , , , , , and . Although they have different representation sizes, , , , and all have in common that they are signed integer types. Likewise , , , and are all unsigned integer types, while , and are distinct in being floating-point types rather than integers. It is common for a piece of code to make sense, for example, only if its arguments are some kind of integer, but not really depend on what particular *kind* of integer. For example, the greatest common denominator algorithm works for all kinds of integers, but will not work for floating-point numbers. Abstract types allow the construction of a hierarchy of types, providing a context into which concrete types can fit. This allows you, for example, to easily program to any type that is an integer, without restricting an algorithm to a specific type of integer.

Abstract types are declared using the keyword. The general syntaxes for declaring an abstract type are:

The keyword introduces a new abstract type, whose name is given by . This name can be optionally followed by and an already-existing type, indicating that the newly declared abstract type is a subtype of this "parent" type.

When no supertype is given, the default supertype is – a predefined abstract type that all objects are instances of and all types are subtypes of. In type theory, is commonly called "top" because it is at the apex of the type graph. Julia also has a predefined abstract "bottom" type, at the nadir of the type graph, which is written as . It is the exact opposite of : no object is an instance of and all types are supertypes of .

Let's consider some of the abstract types that make up Julia's numerical hierarchy:

The type is a direct child type of , and is its child. In turn, has two children (it has more, but only two are shown here; we'll get to the others later): and , separating the world into representations of integers and representations of real numbers. Representations of real numbers include, of course, floating-point types, but also include other types, such as rationals. Hence, is a proper subtype of , including only floating-point representations of real numbers. Integers are further subdivided into and varieties.

The operator in general means "is a subtype of", and, used in declarations like this, declares the right-hand type to be an immediate supertype of the newly declared type. It can also be used in expressions as a subtype operator which returns when its left operand is a subtype of its right operand:

An important use of abstract types is to provide default implementations for concrete types. To give a simple example, consider:

The first thing to note is that the above argument declarations are equivalent to and . When this function is invoked, say as , the dispatcher chooses the most specific method named that matches the given arguments. (See Methods for more information on multiple dispatch.)

Assuming no method more specific than the above is found, Julia next internally defines and compiles a method called specifically for two arguments based on the generic function given above, i.e., it implicitly defines and compiles:

and finally, it invokes this specific method.

Thus, abstract types allow programmers to write generic functions that can later be used as the default method by many combinations of concrete types. Thanks to multiple dispatch, the programmer has full control over whether the default or more specific method is used.

An important point to note is that there is no loss in performance if the programmer relies on a function whose arguments are abstract types, because it is recompiled for each tuple of argument concrete types with which it is invoked. (There may be a performance issue, however, in the case of function arguments that are containers of abstract types; see Performance Tips.)

14.3 Primitive Types

A primitive type is a concrete type whose data consists of plain old bits. Classic examples of primitive types are integers and floating-point values. Unlike most languages, Julia lets you declare your own primitive types, rather than providing only a fixed set of built-in ones. In fact, the standard primitive types are all defined in the language itself:

The general syntaxes for declaring a primitive type are:

The number of bits indicates how much storage the type requires and the name gives the new type a name. A primitive type can optionally be declared to be a subtype of some supertype. If a supertype is omitted, then the type defaults to having as its immediate supertype. The declaration of above therefore means that a boolean value takes eight bits to store, and has as its immediate supertype. Currently, only sizes that are multiples of 8 bits are supported. Therefore, boolean values, although they really need just a single bit, cannot be declared to be any smaller than eight bits.

The types , and all have identical representations: they are eight-bit chunks of memory. Since Julia's type system is nominative, however, they are not interchangeable despite having identical structure. A fundamental difference between them is that they have different supertypes: 's direct supertype is , 's is , and 's is . All other differences between , , and are matters of behavior – the way functions are defined to act when given objects of these types as arguments. This is why a nominative type system is necessary: if structure determined type, which in turn dictates behavior, then it would be impossible to make behave any differently than or .

14.4 Composite Types

Composite types are called records, structs, or objects in various languages. A composite type is a collection of named fields, an instance of which can be treated as a single value. In many languages, composite types are the only kind of user-definable type, and they are by far the most commonly used user-defined type in Julia as well.

In mainstream object oriented languages, such as C++, Java, Python and Ruby, composite types also have named functions associated with them, and the combination is called an "object". In purer object-oriented languages, such as Ruby or Smalltalk, all values are objects whether they are composites or not. In less pure object oriented languages, including C++ and Java, some values, such as integers and floating-point values, are not objects, while instances of user-defined composite types are true objects with associated methods. In Julia, all values are objects, but functions are not bundled with the objects they operate on. This is necessary since Julia chooses which method of a function to use by multiple dispatch, meaning that the types of *all* of a function's arguments are considered when selecting a method, rather than just the first one (see Methods for more information on methods and dispatch). Thus, it would be inappropriate for functions to "belong" to only their first argument. Organizing methods into function objects rather than having named bags of methods "inside" each object ends up being a highly beneficial aspect of the language design.

Composite types are introduced with the keyword followed by a block of field names, optionally annotated with types using the operator:

Fields with no type annotation default to , and can accordingly hold any type of value.

New objects of type are created by applying the type object like a function to values for its fields:

When a type is applied like a function it is called a *constructor*. Two constructors are generated automatically (these are called *default constructors*). One accepts any arguments and calls to convert them to the types of the fields, and the other accepts arguments that match the field types exactly. The reason both of these are generated is that this makes it easier to add new definitions without inadvertently replacing a default constructor.

Since the field is unconstrained in type, any value will do. However, the value for must be convertible to :

You may find a list of field names using the function.

You can access the field values of a composite object using the traditional notation:

Composite objects declared with are *immutable*; they cannot be modified after construction. This may seem odd at first, but it has several advantages:

- It can be more efficient. Some structs can be packed efficiently into arrays, and in some cases the compiler is able to avoid allocating immutable objects entirely.

- It is not possible to violate the invariants provided by the type's constructors.
- Code using immutable objects can be easier to reason about.

An immutable object might contain mutable objects, such as arrays, as fields. Those contained objects will remain mutable; only the fields of the immutable object itself cannot be changed to point to different objects.

Where required, mutable composite objects can be declared with the keyword , to be discussed in the next section.

Composite types with no fields are singletons; there can be only one instance of such types:

The function confirms that the "two" constructed instances of are actually one and the same. Singleton types are described in further detail below.

There is much more to say about how instances of composite types are created, but that discussion depends on both Parametric Types and on Methods, and is sufficiently important to be addressed in its own section: Constructors.

14.5 Mutable Composite Types

If a composite type is declared with instead of , then instances of it can be modified:

In order to support mutation, such objects are generally allocated on the heap, and have stable memory addresses. A mutable object is like a little container that might hold different values over time, and so can only be reliably identified with its address. In contrast, an instance of an immutable type is associated with specific field values -- the field values alone tell you everything about the object. In deciding whether to make a type mutable, ask whether two instances with the same field values would be considered identical, or if they might need to change independently over time. If they would be considered identical, the type should probably be immutable.

To recap, two essential properties define immutability in Julia:

- An object with an immutable type is passed around (both in assignment statements and in function calls) by copying, whereas a mutable type is passed around by reference.
- It is not permitted to modify the fields of a composite immutable type.

It is instructive, particularly for readers whose background is C/C++, to consider why these two properties go hand in hand. If they were separated, i.e., if the fields of objects passed around by copying could be modified, then it would become more difficult to reason about certain instances of generic code. For example, suppose is a function argument of an abstract type, and suppose that the function changes a field: . Depending on whether is passed by copying or by reference, this statement may or may not alter the actual argument in the calling routine. Julia sidesteps the possibility of creating functions with unknown effects in this scenario by forbidding modification of fields of objects passed around by copying.

14.6 Declared Types

The three kinds of types discussed in the previous three sections are actually all closely related. They share the same key properties:

- They are explicitly declared.
- They have names.
- They have explicitly declared supertypes.
- They may have parameters.

Because of these shared properties, these types are internally represented as instances of the same concept, , which is the type of any of these types:

A may be abstract or concrete. If it is concrete, it has a specified size, storage layout, and (optionally) field names. Thus a primitive type is a with nonzero size, but no field names. A composite type is a that has field names or is empty (zero size).

Every concrete value in the system is an instance of some .

14.7 Type Unions

A type union is a special abstract type which includes as objects all instances of any of its argument types, constructed using the special function:

The compilers for many languages have an internal union construct for reasoning about types; Julia simply exposes it to the programmer.

14.8 Parametric Types

An important and powerful feature of Julia's type system is that it is parametric: types can take parameters, so that type declarations actually introduce a whole family of new types – one for each possible combination of parameter values. There are many languages that support some version of generic programming, wherein data structures and algorithms to manipulate them may be specified without specifying the exact types involved. For example, some form of generic programming exists in ML, Haskell, Ada, Eiffel, C++, Java, C#, F#, and Scala, just to name a few. Some of these languages support true parametric polymorphism (e.g. ML, Haskell, Scala), while others support ad-hoc, template-based styles of generic programming (e.g. C++, Java). With so many different varieties of generic programming and parametric types in various languages, we won't even attempt to compare Julia's parametric types to other languages, but will instead focus on explaining Julia's system in its own right. We will note, however, that because Julia is a dynamically typed language and doesn't need to make all type decisions at compile time, many traditional difficulties encountered in static parametric type systems can be relatively easily handled.

All declared types (the variety) can be parameterized, with the same syntax in each case. We will discuss them in the following order: first, parametric composite types, then parametric abstract types, and finally parametric primitive types.

Parametric Composite Types

Type parameters are introduced immediately after the type name, surrounded by curly braces:

This declaration defines a new parametric type, , holding two "coordinates" of type . What, one may ask, is ? Well, that's precisely the point of parametric types: it can be any type at all (or a value of any bits type, actually, although here it's clearly used as a type). is a concrete type equivalent to the type defined by replacing in the definition of with . Thus, this single declaration actually declares an unlimited number of types: , , , etc. Each of these is now a usable concrete type:

The type is a point whose coordinates are 64-bit floating-point values, while the type is a "point" whose "coordinates" are string objects (see Strings).

itself is also a valid type object, containing all instances , , etc. as subtypes:

Other types, of course, are not subtypes of it:

Concrete types with different values of are never subtypes of each other:

> **Warning**
>
> This last point is *very* important: even though we **DO NOT** have .

In other words, in the parlance of type theory, Julia's type parameters are *invariant*, rather than being covariant (or even contravariant). This is for practical reasons: while any instance of may conceptually be like an instance of as well, the two types have different representations in memory:

- An instance of can be represented compactly and efficiently as an immediate pair of 64-bit values;
- An instance of must be able to hold any pair of instances of . Since objects that are instances of can be of arbitrary size and structure, in practice an instance of must be represented as a pair of pointers to individually allocated objects.

The efficiency gained by being able to store objects with immediate values is magnified enormously in the case of arrays: an can be stored as a contiguous memory block of 64-bit floating-point values, whereas an must be an array of pointers to individually allocated objects – which may well be boxed 64-bit floating-point values, but also might be arbitrarily large, complex objects, which are declared to be implementations of the abstract type.

Since is not a subtype of , the following method can't be applied to arguments of type :

A correct way to define a method that accepts all arguments of type where is a subtype of is:

(Equivalently, one could define or ; see UnionAll Types.)

More examples will be discussed later in Methods.

How does one construct a object? It is possible to define custom constructors for composite types, which will be discussed in detail in Constructors, but in the absence of any special constructor declarations, there are two default ways of creating new composite objects, one in which the type parameters are explicitly given and the other in which they are implied by the arguments to the object constructor.

Since the type is a concrete type equivalent to declared with in place of , it can be applied as a constructor accordingly:

For the default constructor, exactly one argument must be supplied for each field:

Only one default constructor is generated for parametric types, since overriding it is not possible. This constructor accepts any arguments and converts them to the field types.

In many cases, it is redundant to provide the type of object one wants to construct, since the types of arguments to the constructor call already implicitly provide type information. For that reason, you can also apply itself as a constructor, provided that the implied value of the parameter type is unambiguous:

In the case of , the type of is unambiguously implied if and only if the two arguments to have the same type. When this isn't the case, the constructor will fail with a :

Constructor methods to appropriately handle such mixed cases can be defined, but that will not be discussed until later on in Constructors.

Parametric Abstract Types

Parametric abstract type declarations declare a collection of abstract types, in much the same way:

With this declaration, is a distinct abstract type for each type or integer value of . As with parametric composite types, each such instance is a subtype of :

Parametric abstract types are invariant, much as parametric composite types are:

The notation can be used to express the Julia analogue of a *covariant* type, while the analogue of a *contravariant* type, but technically these represent *sets* of types (see UnionAll Types).

Much as plain old abstract types serve to create a useful hierarchy of types over concrete types, parametric abstract types serve the same purpose with respect to parametric composite types. We could, for example, have declared to be a subtype of as follows:

Given such a declaration, for each choice of , we have as a subtype of :

This relationship is also invariant:

What purpose do parametric abstract types like serve? Consider if we create a point-like implementation that only requires a single coordinate because the point is on the diagonal line $x = y$:

Now both and are implementations of the abstraction, and similarly for every other possible choice of type . This allows programming to a common interface shared by all objects, implemented for both and . This cannot be fully demonstrated, however, until we have introduced methods and dispatch in the next section, Methods.

There are situations where it may not make sense for type parameters to range freely over all possible types. In such situations, one can constrain the range of like so:

With such a declaration, it is acceptable to use any type that is a subtype of in place of , but not types that are not subtypes of :

Type parameters for parametric composite types can be restricted in the same manner:

To give a real-world example of how all this parametric type machinery can be useful, here is the actual definition of Julia's immutable type (except that we omit the constructor here for simplicity), representing an exact ratio of integers:

It only makes sense to take ratios of integer values, so the parameter type is restricted to being a subtype of , and a ratio of integers represents a value on the real number line, so any is an instance of the abstraction.

Tuple Types

Tuples are an abstraction of the arguments of a function – without the function itself. The salient aspects of a function's arguments are their order and their types. Therefore a tuple type is similar to a parameterized immutable type where each parameter is the type of one field. For example, a 2-element tuple type resembles the following immutable type:

However, there are three key differences:

- Tuple types may have any number of parameters.
- Tuple types are *covariant* in their parameters: is a subtype of . Therefore is considered an abstract type, and tuple types are only concrete if their parameters are.
- Tuples do not have field names; fields are only accessed by index.

Tuple values are written with parentheses and commas. When a tuple is constructed, an appropriate tuple type is generated on demand:

Note the implications of covariance:

Intuitively, this corresponds to the type of a function's arguments being a subtype of the function's signature (when the signature matches).

Vararg Tuple Types

The last parameter of a tuple type can be the special type , which denotes any number of trailing elements:

Notice that corresponds to zero or more elements of type . Vararg tuple types are used to represent the arguments accepted by varargs methods (see Varargs Functions).

The type corresponds to exactly elements of type . is a convenient alias for , i.e. a tuple type containing exactly elements of type .

Singleton Types

There is a special kind of abstract parametric type that must be mentioned here: singleton types. For each type, , the "singleton type" is an abstract type whose only instance is the object . Since the definition is a little difficult to parse, let's look at some examples:

In other words, is true if and only if and are the same object and that object is a type. Without the parameter, is simply an abstract type which has all type objects as its instances, including, of course, singleton types:

Any object that is not a type is not an instance of :

Until we discuss Parametric Methods and conversions, it is difficult to explain the utility of the singleton type construct, but in short, it allows one to specialize function behavior on specific type *values*. This is useful for writing methods

(especially parametric ones) whose behavior depends on a type that is given as an explicit argument rather than implied by the type of one of its arguments.

A few popular languages have singleton types, including Haskell, Scala and Ruby. In general usage, the term ”singleton type” refers to a type whose only instance is a single value. This meaning applies to Julia’s singleton types, but with that caveat that only type objects have singleton types.

Parametric Primitive Types

Primitive types can also be declared parametrically. For example, pointers are represented as primitive types which would be declared in Julia like this:

The slightly odd feature of these declarations as compared to typical parametric composite types, is that the type parameter is not used in the definition of the type itself – it is just an abstract tag, essentially defining an entire family of types with identical structure, differentiated only by their type parameter. Thus, and are distinct types, even though they have identical representations. And of course, all specific pointer types are subtypes of the umbrella type:

14.9 UnionAll Types

We have said that a parametric type like acts as a supertype of all its instances (etc.). How does this work? itself cannot be a normal data type, since without knowing the type of the referenced data the type clearly cannot be used for memory operations. The answer is that (or other parametric types like) is a different kind of type called a type. Such a type expresses the *iterated union* of types for all values of some parameter.

types are usually written using the keyword . For example could be more accurately written as , meaning all values whose type is for some value of . In this context, the parameter is also often called a ”type variable” since it is like a variable that ranges over types. Each introduces a single type variable, so these expressions are nested for types with multiple parameters, for example .

The type application syntax requires to be a type, and first substitutes for the outermost type variable in . The result is expected to be another type, into which is then substituted. So is equivalent to . This explains why it is possible to partially instantiate a type, as in : the first parameter value has been fixed, but the second still ranges over all possible values. Using explicit syntax, any subset of parameters can be fixed. For example, the type of all 1-dimensional arrays can be written as .

Type variables can be restricted with subtype relations. refers to all arrays whose element type is some kind of . The syntax is a convenient shorthand for . Type variables can have both lower and upper bounds. refers to all arrays of s that are able to contain s (since must be at least as big as). The syntax also works to specify only the lower bound of a type variable, and is equivalent to .

Since expressions nest, type variable bounds can refer to outer type variables. For example refers to 2-tuples whose first element is some , and whose second element is an of any kind of array whose element type contains the type of the first tuple element.

The keyword itself can be nested inside a more complex declaration. For example, consider the two types created by the following declarations:

Type defines a 1-dimensional array of 1-dimensional arrays; each of the inner arrays consists of objects of the same type, but this type may vary from one inner array to the next. On the other hand, type defines a 1-dimensional array of 1-dimensional arrays all of whose inner arrays must have the same type. Note that is an abstract type, e.g., , whereas is a concrete type. As a consequence, can be constructed with a zero-argument constructor but cannot.

There is a convenient syntax for naming such types, similar to the short form of function definition syntax:

This is equivalent to . Writing is equivalent to writing , and the umbrella type has as instances all objects where the second parameter – the number of array dimensions – is 1, regardless of what the element type is. In languages where parametric types must always be specified in full, this is not especially helpful, but in Julia, this allows one to write just for the abstract type including all one-dimensional dense arrays of any element type.

14.10 Type Aliases

Sometimes it is convenient to introduce a new name for an already expressible type. This can be done with a simple assignment statement. For example, is aliased to either or as is appropriate for the size of pointers on the system:

This is accomplished via the following code in :

Of course, this depends on what is aliased to – but that is predefined to be the correct type – either or .

(Note that unlike , does not exist as a type alias for a specific sized . Unlike with integer registers, the floating point register sizes are specified by the IEEE-754 standard. Whereas the size of reflects the size of a native pointer on that machine.)

14.11 Operations on Types

Since types in Julia are themselves objects, ordinary functions can operate on them. Some functions that are particularly useful for working with or exploring types have already been introduced, such as the operator, which indicates whether its left hand operand is a subtype of its right hand operand.

The function tests if an object is of a given type and returns true or false:

The function, already used throughout the manual in examples, returns the type of its argument. Since, as noted above, types are objects, they also have types, and we can ask what their types are:

What if we repeat the process? What is the type of a type of a type? As it happens, types are all composite values and thus all have a type of :

 is its own type.

Another operation that applies to some types is , which reveals a type's supertype. Only declared types () have unambiguous supertypes:

If you apply to other type objects (or non-type objects), a is raised:

14.12 Custom pretty-printing

Often, one wants to customize how instances of a type are displayed. This is accomplished by overloading the function. For example, suppose we define a type to represent complex numbers in polar form:

Here, we've added a custom constructor function so that it can take arguments of different types and promote them to a common type (see Constructors and Conversion and Promotion). (Of course, we would have to define lots of other methods, too, to make it act like a , e.g. , , , , promotion rules and so on.) By default, instances of this type display rather simply, with information about the type name and the field values, as e.g. .

If we want it to display instead as , we would define the following method to print the object to a given output object (representing a file, terminal, buffer, etcetera; see Networking and Streams):

More fine-grained control over display of objects is possible. In particular, sometimes one wants both a verbose multi-line printing format, used for displaying a single object in the REPL and other interactive environments, and also a more compact single-line format used for or for displaying the object as part of another object (e.g. in an array). Although by default the function is called in both cases, you can define a *different* multi-line format for displaying an object by overloading a three-argument form of that takes the MIME type as its second argument (see Multimedia I/O), for example:

(Note that here will call the 2-argument method.) This results in:

where the single-line form is still used for an array of values. Technically, the REPL calls to display the result of executing a line, which defaults to , which in turn defaults to , but you should *not* define new methods unless you are defining a new multimedia display handler (see Multimedia I/O).

Moreover, you can also define methods for other MIME types in order to enable richer display (HTML, images, etcetera) of objects in environments that support this (e.g. IJulia). For example, we can define formatted HTML display of objects, with superscripts and italics, via:

A object will then display automatically using HTML in an environment that supports HTML display, but you can call manually to get HTML output if you want:

As a rule of thumb, the single-line method should print a valid Julia expression for creating the shown object. When this method contains infix operators, such as the multiplication operator () in our single-line method for above, it may not parse correctly when printed as part of another object. To see this, consider the expression object (see Program representation) which takes the square of a specific instance of our type:

Because the operator has higher precedence than (see Operator Precedence), this output does not faithfully represent the expression which should be equal to . To solve this issue, we must make a custom method for , which is called internally by the expression object when printing:

The method defined above adds parentheses around the call to when the precedence of the calling operator is higher than or equal to the precedence of multiplication. This check allows expressions which parse correctly without the parentheses (such as and) to omit them when printing:

14.13 "Value types"

In Julia, you can't dispatch on a *value* such as or . However, you can dispatch on parametric types, and Julia allows you to include "plain bits" values (Types, Symbols, Integers, floating-point numbers, tuples, etc.) as type parameters. A common example is the dimensionality parameter in , where is a type (e.g.,) but is just an .

You can create your own custom types that take values as parameters, and use them to control dispatch of custom types. By way of illustration of this idea, let's introduce a parametric type, , and a constructor , which serves as a customary way to exploit this technique for cases where you don't need a more elaborate hierarchy.

 is defined as:

There is no more to the implementation of than this. Some functions in Julia's standard library accept instances as arguments, and you can also use it to write your own functions. For example:

For consistency across Julia, the call site should always pass a *instance* rather than using a *type*, i.e., use rather than .

It's worth noting that it's extremely easy to mis-use parametric "value" types, including ; in unfavorable cases, you can easily end up making the performance of your code much *worse*. In particular, you would never want to write actual code as illustrated above. For more information about the proper (and improper) uses of , please read the more extensive discussion in the performance tips.

14.14 Nullable Types: Representing Missing Values

In many settings, you need to interact with a value of type that may or may not exist. To handle these settings, Julia provides a parametric type called , which can be thought of as a specialized container type that can contain either zero or one values. provides a minimal interface designed to ensure that interactions with missing values are safe. At present, the interface consists of several possible interactions:

- Construct a object.
- Check if a object has a missing value.
- Access the value of a object with a guarantee that a will be thrown if the object's value is missing.

- Access the value of a object with a guarantee that a default value of type will be returned if the object's value is missing.
- Perform an operation on the value (if it exists) of a object, getting a result. The result will be missing if the original value was missing.
- Performing a test on the value (if it exists) of a object, getting a result that is missing if either the itself was missing, or the test failed.
- Perform general operations on single objects, propagating the missing data.

Constructing objects

To construct an object representing a missing value of type , use the function:

To construct an object representing a non-missing value of type , use the function:

Note the core distinction between these two ways of constructing a object: in one style, you provide a type, , as a function parameter; in the other style, you provide a single value of type as an argument.

Checking if a object has a value

You can check if a object has any value using :

Safely accessing the value of a object

You can safely access the value of a object using :

If the value is not present, as it would be for , a error will be thrown. The error-throwing nature of the function ensures that any attempt to access a missing value immediately fails.

In cases for which a reasonable default value exists that could be used when a object's value turns out to be missing, you can provide this default value as a second argument to :

Tip

Make sure the type of the default value passed to and that of the object match to avoid type instability, which could hurt performance. Use manually if needed.

Performing operations on objects

objects represent values that are possibly missing, and it is possible to write all code using these objects by first testing to see if the value is missing with , and then doing an appropriate action. However, there are some common use cases where the code could be more concise or clear by using a higher-order function.

The function takes as arguments a function and a value . It produces a :

- If is a missing value, then it produces a missing value;
- If has a value, then it produces a containing as value.

This is useful for performing simple operations on values that might be missing if the desired behaviour is to simply propagate the missing values forward.

The function takes as arguments a predicate function (that is, a function returning a boolean) and a value . It produces a value:

- If is a missing value, then it produces a missing value;
- If is true, then it produces the original value ;
- If is false, then it produces a missing value.

In this way, can be thought of as selecting only allowable values, and converting non-allowable values to missing values.

While and are useful in specific cases, by far the most useful higher-order function is , which can handle a wide variety of cases, including making existing operations work and propagate s. An example will motivate the need for . Suppose we have a function that computes the greater of two real roots of a quadratic equation, using the quadratic formula:

We may verify that the result of is , as we expect, since is the greater of two real roots of the quadratic equation.

Suppose now that we want to find the greatest real root of a quadratic equations where the coefficients might be missing values. Having missing values in datasets is a common occurrence in real-world data, and so it is important to be able to deal with them. But we cannot find the roots of an equation if we do not know all the coefficients. The best solution to this will depend on the particular use case; perhaps we should throw an error. However, for this example, we will assume that the best solution is to propagate the missing values forward; that is, if any input is missing, we simply produce a missing output.

The function makes this task easy; we can simply pass the function we wrote to :

If one or more of the inputs is missing, then the output of will be missing.

There exists special syntactic sugar for the function using a dot notation:

In particular, the regular arithmetic operators can be conveniently using -prefixed operators:

Chapter 15

Methods

Recall from Functions that a function is an object that maps a tuple of arguments to a return value, or throws an exception if no appropriate value can be returned. It is common for the same conceptual function or operation to be implemented quite differently for different types of arguments: adding two integers is very different from adding two floating-point numbers, both of which are distinct from adding an integer to a floating-point number. Despite their implementation differences, these operations all fall under the general concept of "addition". Accordingly, in Julia, these behaviors all belong to a single object: the function.

To facilitate using many different implementations of the same concept smoothly, functions need not be defined all at once, but can rather be defined piecewise by providing specific behaviors for certain combinations of argument types and counts. A definition of one possible behavior for a function is called a *method*. Thus far, we have presented only examples of functions defined with a single method, applicable to all types of arguments. However, the signatures of method definitions can be annotated to indicate the types of arguments in addition to their number, and more than a single method definition may be provided. When a function is applied to a particular tuple of arguments, the most specific method applicable to those arguments is applied. Thus, the overall behavior of a function is a patchwork of the behaviors of its various method definitions. If the patchwork is well designed, even though the implementations of the methods may be quite different, the outward behavior of the function will appear seamless and consistent.

The choice of which method to execute when a function is applied is called *dispatch*. Julia allows the dispatch process to choose which of a function's methods to call based on the number of arguments given, and on the types of all of the function's arguments. This is different than traditional object-oriented languages, where dispatch occurs based only on the first argument, which often has a special argument syntax, and is sometimes implied rather than explicitly written as an argument. [1] Using all of a function's arguments to choose which method should be invoked, rather than just the first, is known as multiple dispatch. Multiple dispatch is particularly useful for mathematical code, where it makes little sense to artificially deem the operations to "belong" to one argument more than any of the others: does the addition operation in belong to any more than it does to ? The implementation of a mathematical operator generally depends on the types of all of its arguments. Even beyond mathematical operations, however, multiple dispatch ends up being a powerful and convenient paradigm for structuring and organizing programs.

15.1 Defining Methods

Until now, we have, in our examples, defined only functions with a single method having unconstrained argument types. Such functions behave just like they would in traditional dynamically typed languages. Nevertheless, we have used multiple dispatch and methods almost continually without being aware of it: all of Julia's standard functions and operators,

[1] In C++ or Java, for example, in a method call like , the object obj "receives" the method call and is implicitly passed to the method via the keyword, rather than as an explicit method argument. When the current object is the receiver of a method call, it can be omitted altogether, writing just , with implied as the receiving object.

like the aforementioned function, have many methods defining their behavior over various possible combinations of argument type and count.

When defining a function, one can optionally constrain the types of parameters it is applicable to, using the type-assertion operator, introduced in the section on Composite Types:

This function definition applies only to calls where and are both values of type :

Applying it to any other types of arguments will result in a :

As you can see, the arguments must be precisely of type . Other numeric types, such as integers or 32-bit floating-point values, are not automatically converted to 64-bit floating-point, nor are strings parsed as numbers. Because is a concrete type and concrete types cannot be subclassed in Julia, such a definition can only be applied to arguments that are exactly of type . It may often be useful, however, to write more general methods where the declared parameter types are abstract:

This method definition applies to any pair of arguments that are instances of . They need not be of the same type, so long as they are each numeric values. The problem of handling disparate numeric types is delegated to the arithmetic operations in the expression .

To define a function with multiple methods, one simply defines the function multiple times, with different numbers and types of arguments. The first method definition for a function creates the function object, and subsequent method

definitions add new methods to the existing function object. The most specific method definition matching the number and types of the arguments will be executed when the function is applied. Thus, the two method definitions above, taken together, define the behavior for over all pairs of instances of the abstract type – but with a different behavior specific to pairs of values. If one of the arguments is a 64-bit float but the other one is not, then the method cannot be called and the more general method must be used:

The definition is only used in the first case, while the definition is used in the others. No automatic casting or conversion of function arguments is ever performed: all conversion in Julia is non-magical and completely explicit. Conversion and Promotion, however, shows how clever application of sufficiently advanced technology can be indistinguishable from magic. [2]

For non-numeric values, and for fewer or more than two arguments, the function remains undefined, and applying it will still result in a :

You can easily see which methods exist for a function by entering the function object itself in an interactive session:

This output tells us that is a function object with two methods. To find out what the signatures of those methods are, use the function:

which shows that has two methods, one taking two arguments and one taking arguments of type . It also indicates the file and line number where the methods were defined: because these methods were defined at the REPL, we get the apparent line number .

In the absence of a type declaration with , the type of a method parameter is by default, meaning that it is unconstrained since all values in Julia are instances of the abstract type . Thus, we can define a catch-all method for like so:

This catch-all is less specific than any other possible method definition for a pair of parameter values, so it will only be called on pairs of arguments to which no other method definition applies.

Although it seems a simple concept, multiple dispatch on the types of values is perhaps the single most powerful and central feature of the Julia language. Core operations typically have dozens of methods:

Multiple dispatch together with the flexible parametric type system give Julia its ability to abstractly express high-level algorithms decoupled from implementation details, yet generate efficient, specialized code to handle each case at run time.

15.2 Method Ambiguities

It is possible to define a set of function methods such that there is no unique most specific method applicable to some combinations of arguments:

Here the call could be handled by either the or the method, and neither is more specific than the other. In such cases, Julia raises a rather than arbitrarily picking a method. You can avoid method ambiguities by specifying an appropriate method for the intersection case:

It is recommended that the disambiguating method be defined first, since otherwise the ambiguity exists, if transiently, until the more specific method is defined.

In more complex cases, resolving method ambiguities involves a certain element of design; this topic is explored further below.

15.3 Parametric Methods

Method definitions can optionally have type parameters qualifying the signature:

The first method applies whenever both arguments are of the same concrete type, regardless of what type that is, while the second method acts as a catch-all, covering all other cases. Thus, overall, this defines a boolean function that checks whether its two arguments are of the same type:

Such definitions correspond to methods whose type signatures are types (see UnionAll Types).

This kind of definition of function behavior by dispatch is quite common – idiomatic, even – in Julia. Method type parameters are not restricted to being used as the types of arguments: they can be used anywhere a value would be in the signature of the function or body of the function. Here's an example where the method type parameter is used as the type parameter to the parametric type in the method signature:

As you can see, the type of the appended element must match the element type of the vector it is appended to, or else a is raised. In the following example, the method type parameter is used as the return value:

Just as you can put subtype constraints on type parameters in type declarations (see Parametric Types), you can also constrain type parameters of methods:

The function behaves much like the function defined above, but is only defined for pairs of numbers.

Parametric methods allow the same syntax as expressions used to write types (see UnionAll Types). If there is only a single parameter, the enclosing curly braces (in) can be omitted, but are often preferred for clarity. Multiple parameters can be separated with commas, e.g. , or written using nested , e.g. .

15.4 Redefining Methods

When redefining a method or adding new methods, it is important to realize that these changes don't take effect immediately. This is key to Julia's ability to statically infer and compile code to run fast, without the usual JIT tricks and overhead. Indeed, any new method definition won't be visible to the current runtime environment, including Tasks and Threads (and any previously defined functions). Let's start with an example to see what this means:

In this example, observe that the new definition for has been created, but can't be immediately called. The new global is immediately visible to the function, so you could write (without parentheses). But neither you, nor any of your callers, nor the functions they call, or etc. can call this new method definition!

But there's an exception: future calls to *from the REPL* work as expected, being able to both see and call the new definition of .

However, future calls to will continue to see the definition of as it was *at the previous statement at the REPL*, and thus before that call to .

You may want to try this for yourself to see how it works.

The implementation of this behavior is a "world age counter". This monotonically increasing value tracks each method definition operation. This allows describing "the set of method definitions visible to a given runtime environment" as a single number, or "world age". It also allows comparing the methods available in two worlds just by comparing their ordinal value. In the example above, we see that the "current world" (in which the method exists), is one greater than the task-local "runtime world" that was fixed when the execution of started.

Sometimes it is necessary to get around this (for example, if you are implementing the above REPL). Fortunately, there is an easy solution: call the function using :

Finally, let's take a look at some more complex examples where this rule comes into play. Define a function , which initially has one method:

Start some other operations that use :

Now we add some new methods to :

Compare how these results differ:

15.5 Parametrically-constrained Varargs methods

Function parameters can also be used to constrain the number of arguments that may be supplied to a ”varargs” function (Varargs Functions). The notation is used to indicate such a constraint. For example:

More usefully, it is possible to constrain varargs methods by a parameter. For example:

would be called only when the number of matches the dimensionality of the array.

When only the type of supplied arguments needs to be constrained can be equivalently written as . For instance is a shorthand for .

15.6 Note on Optional and keyword Arguments

As mentioned briefly in Functions, optional arguments are implemented as syntax for multiple method definitions. For example, this definition:

translates to the following three methods:

This means that calling is equivalent to calling . In this case the result is , because invokes the first method of above. However, this need not always be the case. If you define a fourth method that is more specialized for integers:

then the result of both and is . In other words, optional arguments are tied to a function, not to any specific method of that function. It depends on the types of the optional arguments which method is invoked. When optional arguments are defined in terms of a global variable, the type of the optional argument may even change at run-time.

Keyword arguments behave quite differently from ordinary positional arguments. In particular, they do not participate in method dispatch. Methods are dispatched based only on positional arguments, with keyword arguments processed after the matching method is identified.

15.7 Function-like objects

Methods are associated with types, so it is possible to make any arbitrary Julia object ”callable” by adding methods to its type. (Such ”callable” objects are sometimes called ”functors.”)

For example, you can define a type that stores the coefficients of a polynomial, but behaves like a function evaluating the polynomial:

Notice that the function is specified by type instead of by name. In the function body, will refer to the object that was called. A can be used as follows:

This mechanism is also the key to how type constructors and closures (inner functions that refer to their surrounding environment) work in Julia, discussed later in the manual.

15.8 Empty generic functions

Occasionally it is useful to introduce a generic function without yet adding methods. This can be used to separate interface definitions from implementations. It might also be done for the purpose of documentation or code readability. The syntax for this is an empty block without a tuple of arguments:

15.9 Method design and the avoidance of ambiguities

Julia's method polymorphism is one of its most powerful features, yet exploiting this power can pose design challenges. In particular, in more complex method hierarchies it is not uncommon for ambiguities to arise.

Above, it was pointed out that one can resolve ambiguities like

by defining a method

This is often the right strategy; however, there are circumstances where following this advice blindly can be counterproductive. In particular, the more methods a generic function has, the more possibilities there are for ambiguities. When your method hierarchies get more complicated than this simple example, it can be worth your while to think carefully about alternative strategies.

Below we discuss particular challenges and some alternative ways to resolve such issues.

Tuple and NTuple arguments

(and) arguments present special challenges. For example,

are ambiguous because of the possibility that : there are no elements to determine whether the or variant should be called. To resolve the ambiguity, one approach is define a method for the empty tuple:

Alternatively, for all methods but one you can insist that there is at least one element in the tuple:

Orthogonalize your design

When you might be tempted to dispatch on two or more arguments, consider whether a ”wrapper” function might make for a simpler design. For example, instead of writing multiple variants:

you might consider defining

where converts the argument to type . This is a very specific example of the more general principle of orthogonal design, in which separate concepts are assigned to separate methods. Here, will most likely need a fallback definition

A related strategy exploits to bring and to a common type:

One risk with this design is the possibility that if there is no suitable promotion method converting and to the same type, the second method will recurse on itself infinitely and trigger a stack overflow. The non-exported function can be used as an alternative; when promotion fails it will still throw an error, but one that fails faster with a more specific error message.

Dispatch on one argument at a time

If you need to dispatch on multiple arguments, and there are many fallbacks with too many combinations to make it practical to define all possible variants, then consider introducing a ”name cascade” where (for example) you dispatch on the first argument and then call an internal method:

Then the internal methods and can dispatch on without concern about ambiguities with each other with respect to .

Be aware that this strategy has at least one major disadvantage: in many cases, it is not possible for users to further customize the behavior of by defining further specializations of your exported function . Instead, they have to define specializations for your internal methods and , and this blurs the lines between exported and internal methods.

Abstract containers and element types

Where possible, try to avoid defining methods that dispatch on specific element types of abstract containers. For example,

generates ambiguities for anyone who defines a method

The best approach is to avoid defining *either* of these methods: instead, rely on a generic method and make sure this method is implemented with generic calls (like and) that do the right thing for each container type and element type *separately*. This is just a more complex variant of the advice to orthogonalize your methods.

When this approach is not possible, it may be worth starting a discussion with other developers about resolving the ambiguity; just because one method was defined first does not necessarily mean that it can't be modified or eliminated. As a last resort, one developer can define the "band-aid" method

that resolves the ambiguity by brute force.

Complex method "cascades" with default arguments

If you are defining a method "cascade" that supplies defaults, be careful about dropping any arguments that correspond to potential defaults. For example, suppose you're writing a digital filtering algorithm and you have a method that handles the edges of the signal by applying padding:

This will run afoul of a method that supplies default padding:

Together, these two methods generate an infinite recursion with constantly growing bigger.

The better design would be to define your call hierarchy like this:

is supplied in the same argument position as any other kind of padding, so it keeps the dispatch hierarchy well organized and with reduced likelihood of ambiguities. Moreover, it extends the ”public” interface: a user who wants to control the padding explicitly can call the variant directly.

[2] Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.

Chapter 16

Constructors

Constructors [1] are functions that create new objects – specifically, instances of Composite Types. In Julia, type objects also serve as constructor functions: they create new instances of themselves when applied to an argument tuple as a function. This much was already mentioned briefly when composite types were introduced. For example:

For many types, forming new objects by binding their field values together is all that is ever needed to create instances. There are, however, cases where more functionality is required when creating composite objects. Sometimes invariants must be enforced, either by checking arguments or by transforming them. Recursive data structures, especially those that may be self-referential, often cannot be constructed cleanly without first being created in an incomplete state and then altered programmatically to be made whole, as a separate step from object creation. Sometimes, it's just convenient to be able to construct objects with fewer or different types of parameters than they have fields. Julia's system for object construction addresses all of these cases and more.

16.1 Outer Constructor Methods

A constructor is just like any other function in Julia in that its overall behavior is defined by the combined behavior of its methods. Accordingly, you can add functionality to a constructor by simply defining new methods. For example, let's say you want to add a constructor method for objects that takes only one argument and uses the given value for both the and fields. This is simple:

[1] Nomenclature: while the term "constructor" generally refers to the entire function which constructs objects of a type, it is common to abuse terminology slightly and refer to specific constructor methods as "constructors". In such situations, it is generally clear from context that the term is used to mean "constructor method" rather than "constructor function", especially as it is often used in the sense of singling out a particular method of the constructor from all of the others.

You could also add a zero-argument constructor method that supplies default values for both of the and fields:

Here the zero-argument constructor method calls the single-argument constructor method, which in turn calls the automatically provided two-argument constructor method. For reasons that will become clear very shortly, additional constructor methods declared as normal methods like this are called *outer* constructor methods. Outer constructor methods can only ever create a new instance by calling another constructor method, such as the automatically provided default ones.

16.2 Inner Constructor Methods

While outer constructor methods succeed in addressing the problem of providing additional convenience methods for constructing objects, they fail to address the other two use cases mentioned in the introduction of this chapter: enforcing invariants, and allowing construction of self-referential objects. For these problems, one needs *inner* constructor methods. An inner constructor method is much like an outer constructor method, with two differences:

1. It is declared inside the block of a type declaration, rather than outside of it like normal methods.
2. It has access to a special locally existent function called that creates objects of the block's type.

For example, suppose one wants to declare a type that holds a pair of real numbers, subject to the constraint that the first number is not greater than the second one. One could declare it like this:

Now objects can only be constructed such that :

If the type were declared , you could reach in and directly change the field values to violate this invariant, but messing around with an object's internals uninvited is considered poor form. You (or someone else) can also provide additional outer constructor methods at any later point, but once a type is declared, there is no way to add more inner constructor methods. Since outer constructor methods can only create objects by calling other constructor methods, ultimately, some inner constructor must be called to create an object. This guarantees that all objects of the declared type must come into existence by a call to one of the inner constructor methods provided with the type, thereby giving some degree of enforcement of a type's invariants.

If any inner constructor method is defined, no default constructor method is provided: it is presumed that you have supplied yourself with all the inner constructors you need. The default constructor is equivalent to writing your own inner constructor method that takes all of the object's fields as parameters (constrained to be of the correct type, if the corresponding field has a type), and passes them to , returning the resulting object:

This declaration has the same effect as the earlier definition of the type without an explicit inner constructor method. The following two types are equivalent – one with a default constructor, the other with an explicit constructor:

It is considered good form to provide as few inner constructor methods as possible: only those taking all arguments explicitly and enforcing essential error checking and transformation. Additional convenience constructor methods, supplying default values or auxiliary transformations, should be provided as outer constructors that call the inner constructors to do the heavy lifting. This separation is typically quite natural.

16.3 Incomplete Initialization

The final problem which has still not been addressed is construction of self-referential objects, or more generally, recursive data structures. Since the fundamental difficulty may not be immediately obvious, let us briefly explain it. Consider the following recursive type declaration:

This type may appear innocuous enough, until one considers how to construct an instance of it. If is an instance of , then a second instance can be created by the call:

But how does one construct the first instance when no instance exists to provide as a valid value for its field? The only solution is to allow creating an incompletely initialized instance of with an unassigned field, and using that incomplete instance as a valid value for the field of another instance, such as, for example, itself.

To allow for the creation of incompletely initialized objects, Julia allows the function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized. The inner constructor method can then use the incomplete object, finishing its initialization before returning it. Here, for example, we take another crack at defining the type, with a zero-argument inner constructor returning instances having fields pointing to themselves:

We can verify that this constructor works and constructs objects that are, in fact, self-referential:

Although it is generally a good idea to return a fully initialized object from an inner constructor, incompletely initialized objects can be returned:

While you are allowed to create objects with uninitialized fields, any access to an uninitialized reference is an immediate error:

This avoids the need to continually check for values. However, not all object fields are references. Julia considers some types to be "plain data", meaning all of their data is self-contained and does not reference other objects. The plain data types consist of primitive types (e.g.) and immutable structs of other plain data types. The initial contents of a plain data type is undefined:

Arrays of plain data types exhibit the same behavior.

You can pass incomplete objects to other functions from inner constructors to delegate their completion:

As with incomplete objects returned from constructors, if or any of its callees try to access the field of the object before it has been initialized, an error will be thrown immediately.

16.4 Parametric Constructors

Parametric types add a few wrinkles to the constructor story. Recall from Parametric Types that, by default, instances of parametric composite types can be constructed either with explicitly given type parameters or with type parameters implied by the types of the arguments given to the constructor. Here are some examples:

As you can see, for constructor calls with explicit type parameters, the arguments are converted to the implied field types: works, but raises an when converting to . When the type is implied by the arguments to the constructor call, as in , then the types of the arguments must agree – otherwise the cannot be determined – but any pair of real arguments with matching type may be given to the generic constructor.

What's really going on here is that , and are all different constructor functions. In fact, is a distinct constructor function for each type . Without any explicitly provided inner constructors, the declaration of the composite type automatically provides an inner constructor, , for each possible type , that behaves just like non-parametric default inner constructors do. It also provides a single general outer constructor that takes pairs of real arguments, which must be of the same type. This automatic provision of constructors is equivalent to the following explicit declaration:

Notice that each definition looks like the form of constructor call that it handles. The call will invoke the definition inside the block. The outer constructor declaration, on the other hand, defines a method for the general constructor which only applies to pairs of values of the same real type. This declaration makes constructor calls without explicit type parameters, like and , work. Since the method declaration restricts the arguments to being of the same type, calls like , with arguments of different types, result in "no method" errors.

Suppose we wanted to make the constructor call work by "promoting" the integer value to the floating-point value . The simplest way to achieve this is to define the following additional outer constructor method:

This method uses the function to explicitly convert to and then delegates construction to the general constructor for the case where both arguments are . With this method definition what was previously a now successfully creates a point of type :

However, other similar calls still don't work:

For a more general way to make all such calls work sensibly, see Conversion and Promotion. At the risk of spoiling the suspense, we can reveal here that all it takes is the following outer method definition to make all calls to the general constructor work as one would expect:

The function converts all its arguments to a common type – in this case . With this method definition, the constructor promotes its arguments the same way that numeric operators like do, and works for all kinds of real numbers:

Thus, while the implicit type parameter constructors provided by default in Julia are fairly strict, it is possible to make them behave in a more relaxed but sensible manner quite easily. Moreover, since constructors can leverage all of the power of the type system, methods, and multiple dispatch, defining sophisticated behavior is typically quite simple.

16.5 Case Study: Rational

Perhaps the best way to tie all these pieces together is to present a real world example of a parametric composite type and its constructor methods. To that end, here is the (slightly modified) beginning of , which implements Julia's Rational Numbers:

The first line – – declares that takes one type parameter of an integer type, and is itself a real type. The field declarations and indicate that the data held in a object are a pair of integers of type , one representing the rational value's numerator and the other representing its denominator.

Now things get interesting. has a single inner constructor method which checks that both of and aren't zero and ensures that every rational is constructed in "lowest terms" with a non-negative denominator. This is accomplished by dividing the given numerator and denominator values by their greatest common divisor, computed using the function. Since returns the greatest common divisor of its arguments with sign matching the first argument (here), after this division the new value of is guaranteed to be non-negative. Because this is the only inner constructor for , we can be certain that objects are always constructed in this normalized form.

also provides several outer constructor methods for convenience. The first is the "standard" general constructor that infers the type parameter from the type of the numerator and denominator when they have the same type. The second applies when the given numerator and denominator values have different types: it promotes them to a common type and then delegates construction to the outer constructor for arguments of matching type. The third outer constructor turns integer values into rationals by supplying a value of as the denominator.

Following the outer constructor definitions, we have a number of methods for the operator, which provides a syntax for writing rationals. Before these definitions, is a completely undefined operator with only syntax and no meaning. Afterwards, it behaves just as described in Rational Numbers – its entire behavior is defined in these few lines. The first and most basic definition just makes construct a by applying the constructor to and when they are integers. When one of the operands of is already a rational number, we construct a new rational for the resulting ratio slightly differently; this behavior is actually identical to division of a rational with an integer. Finally, applying to complex integral values creates an instance of – a complex number whose real and imaginary parts are rationals:

Thus, although the operator usually returns an instance of , if either of its arguments are complex integers, it will return an instance of instead. The interested reader should consider perusing the rest of : it is short, self-contained, and implements an entire basic Julia type.

16.6 Constructors and Conversion

Constructors in Julia are implemented like other callable objects: methods are added to their types. The type of a type is , so all constructor methods are stored in the method table for the type. This means that you can declare more flexible constructors, e.g. constructors for abstract types, by explicitly defining methods for the appropriate types.

However, in some cases you could consider adding methods to *instead* of defining a constructor, because Julia falls back to calling if no matching constructor is found. For example, if no constructor exists is called.

is used extensively throughout Julia whenever one type needs to be converted to another (e.g. in assignment, , etcetera), and should generally only be defined (or successful) if the conversion is lossless. For example, produces , but throws an . If you want to define a constructor for a lossless conversion from one type to another, you should probably define a method instead.

On the other hand, if your constructor does not represent a lossless conversion, or doesn't represent "conversion" at all, it is better to leave it as a constructor rather than a method. For example, the constructor creates a zero-dimensional of the type , but is not really a "conversion" from to an .

16.7 Outer-only constructors

As we have seen, a typical parametric type has inner constructors that are called when type parameters are known; e.g. they apply to but not to . Optionally, outer constructors that determine type parameters automatically can be added, for example constructing a from the call . Outer constructors call inner constructors to do the core work of making an instance. However, in some cases one would rather not provide inner constructors, so that specific type parameters cannot be requested manually.

For example, say we define a type that stores a vector along with an accurate representation of its sum:

The problem is that we want to be a larger type than , so that we can sum many elements with less information loss. For example, when is , we would like to be . Therefore we want to avoid an interface that allows the user to construct instances of the type . One way to do this is to provide a constructor only for , but inside the definition block to suppress generation of default constructors:

This constructor will be invoked by the syntax . The syntax allows specifying parameters for the type to be constructed, i.e. this call will return a . can be used in any constructor definition, but for convenience the parameters to are automatically derived from the type being constructed when possible.

Chapter 17

Conversion and Promotion

Julia has a system for promoting arguments of mathematical operators to a common type, which has been mentioned in various other sections, including Integers and Floating-Point Numbers, Mathematical Operations and Elementary Functions, Types, and Methods. In this section, we explain how this promotion system works, as well as how to extend it to new types and apply it to functions besides built-in mathematical operators. Traditionally, programming languages fall into two camps with respect to promotion of arithmetic arguments:

- **Automatic promotion for built-in arithmetic types and operators.** In most languages, built-in numeric types, when used as operands to arithmetic operators with infix syntax, such as , , , and , are automatically promoted to a common type to produce the expected results. C, Java, Perl, and Python, to name a few, all correctly compute the sum as the floating-point value , even though one of the operands to is an integer. These systems are convenient and designed carefully enough that they are generally all-but-invisible to the programmer: hardly anyone consciously thinks of this promotion taking place when writing such an expression, but compilers and interpreters must perform conversion before addition since integers and floating-point values cannot be added as-is. Complex rules for such automatic conversions are thus inevitably part of specifications and implementations for such languages.

- **No automatic promotion.** This camp includes Ada and ML – very ”strict” statically typed languages. In these languages, every conversion must be explicitly specified by the programmer. Thus, the example expression would be a compilation error in both Ada and ML. Instead one must write , explicitly converting the integer to a floating-point value before performing addition. Explicit conversion everywhere is so inconvenient, however, that even Ada has some degree of automatic conversion: integer literals are promoted to the expected integer type automatically, and floating-point literals are similarly promoted to appropriate floating-point types.

In a sense, Julia falls into the ”no automatic promotion” category: mathematical operators are just functions with special syntax, and the arguments of functions are never automatically converted. However, one may observe that applying mathematical operations to a wide variety of mixed argument types is just an extreme case of polymorphic multiple dispatch – something which Julia's dispatch and type systems are particularly well-suited to handle. ”Automatic” promotion of mathematical operands simply emerges as a special application: Julia comes with pre-defined catch-all dispatch rules for mathematical operators, invoked when no specific implementation exists for some combination of operand types. These catch-all rules first promote all operands to a common type using user-definable promotion rules, and then invoke a specialized implementation of the operator in question for the resulting values, now of the same type. User-defined types can easily participate in this promotion system by defining methods for conversion to and from other types, and providing a handful of promotion rules defining what types they should promote to when mixed with other types.

17.1 Conversion

Conversion of values to various types is performed by the function. The function generally takes two arguments: the first is a type object while the second is a value to convert to that type; the returned value is the value converted to an instance of given type. The simplest way to understand this function is to see it in action:

Conversion isn't always possible, in which case a no method error is thrown indicating that doesn't know how to perform the requested conversion:

Some languages consider parsing strings as numbers or formatting numbers as strings to be conversions (many dynamic languages will even perform conversion for you automatically), however Julia does not: even though some strings can be parsed as numbers, most strings are not valid representations of numbers, and only a very limited subset of them are. Therefore in Julia the dedicated function must be used to perform this operation, making it more explicit.

Defining New Conversions

To define a new conversion, simply provide a new method for . That's really all there is to it. For example, the method to convert a real number to a boolean is this:

The type of the first argument of this method is a singleton type, , the only instance of which is . Thus, this method is only invoked when the first argument is the type value . Notice the syntax used for the first argument: the argument name is omitted prior to the symbol, and only the type is given. This is the syntax in Julia for a function argument whose type is specified but whose value is never used in the function body. In this example, since the type is a singleton, there would never be any reason to use its value within the body. When invoked, the method determines whether a numeric value is true or false as a boolean, by comparing it to one and zero:

The method signatures for conversion methods are often quite a bit more involved than this example, especially for parametric types. The example above is meant to be pedagogical, and is not the actual Julia behaviour. This is the actual implementation in Julia:

Case Study: Rational Conversions

To continue our case study of Julia's type, here are the conversions declared in , right after the declaration of the type and its constructors:

The initial four convert methods provide conversions to rational types. The first method converts one type of rational to another type of rational by converting the numerator and denominator to the appropriate integer type. The second method does the same conversion for integers by taking the denominator to be 1. The third method implements a standard algorithm for approximating a floating-point number by a ratio of integers to within a given tolerance, and the fourth method applies it, using machine epsilon at the given value as the threshold. In general, one should have .

The last two convert methods provide conversions from rational types to floating-point and integer types. To convert to floating point, one simply converts both numerator and denominator to that floating point type and then divides. To convert to integer, one can use the operator for truncated integer division (rounded towards zero).

17.2 Promotion

Promotion refers to converting values of mixed types to a single common type. Although it is not strictly necessary, it is generally implied that the common type to which the values are converted can faithfully represent all of the original values. In this sense, the term ”promotion” is appropriate since the values are converted to a ”greater” type – i.e. one which can represent all of the input values in a single common type. It is important, however, not to confuse this with object-oriented (structural) super-typing, or Julia's notion of abstract super-types: promotion has nothing to do with the type hierarchy, and everything to do with converting between alternate representations. For instance, although every value can also be represented as a value, is not a subtype of .

Promotion to a common ”greater” type is performed in Julia by the function, which takes any number of arguments, and returns a tuple of the same number of values, converted to a common type, or throws an exception if promotion is not possible. The most common use case for promotion is to convert numeric arguments to a common type:

Floating-point values are promoted to the largest of the floating-point argument types. Integer values are promoted to the larger of either the native machine word size or the largest integer argument type. Mixtures of integers and floating-point values are promoted to a floating-point type big enough to hold all the values. Integers mixed with rationals are promoted to rationals. Rationals mixed with floats are promoted to floats. Complex values mixed with real values are promoted to the appropriate kind of complex value.

That is really all there is to using promotions. The rest is just a matter of clever application, the most typical ”clever” application being the definition of catch-all methods for numeric operations like the arithmetic operators , , and . Here are some of the catch-all method definitions given in :

These method definitions say that in the absence of more specific rules for adding, subtracting, multiplying and dividing pairs of numeric values, promote the values to a common type and then try again. That's all there is to it: nowhere else does one ever need to worry about promotion to a common numeric type for arithmetic operations – it just happens automatically. There are definitions of catch-all promotion methods for a number of other arithmetic and mathematical functions in , but beyond that, there are hardly any calls to required in the Julia standard library. The most common usages of occur in outer constructors methods, provided for convenience, to allow constructor calls with mixed types to delegate to an inner type with fields promoted to an appropriate common type. For example, recall that provides the following outer constructor method:

This allows calls like the following to work:

For most user-defined types, it is better practice to require programmers to supply the expected types to constructor functions explicitly, but sometimes, especially for numeric problems, it can be convenient to do promotion automatically.

Defining Promotion Rules

Although one could, in principle, define methods for the function directly, this would require many redundant definitions for all possible permutations of argument types. Instead, the behavior of is defined in terms of an auxiliary function called , which one can provide methods for. The function takes a pair of type objects and returns another type object, such that instances of the argument types will be promoted to the returned type. Thus, by defining the rule:

one declares that when 64-bit and 32-bit floating-point values are promoted together, they should be promoted to 64-bit floating-point. The promotion type does not need to be one of the argument types, however; the following promotion rules both occur in Julia's standard library:

In the latter case, the result type is since is the only type large enough to hold integers for arbitrary-precision integer arithmetic. Also note that one does not need to define both and – the symmetry is implied by the way is used in the promotion process.

The function is used as a building block to define a second function called , which, given any number of type objects, returns the common type to which those values, as arguments to should be promoted. Thus, if one wants to know, in absence of actual values, what type a collection of values of certain types would promote to, one can use :

Internally, is used inside of to determine what type argument values should be converted to for promotion. It can, however, be useful in its own right. The curious reader can read the code in , which defines the complete promotion mechanism in about 35 lines.

Case Study: Rational Promotions

Finally, we finish off our ongoing case study of Julia's rational number type, which makes relatively sophisticated use of the promotion mechanism with the following promotion rules:

The first rule says that promoting a rational number with any other integer type promotes to a rational type whose numerator/denominator type is the result of promotion of its numerator/denominator type with the other integer type. The second rule applies the same logic to two different types of rational numbers, resulting in a rational of the promotion of their respective numerator/denominator types. The third and final rule dictates that promoting a rational with a float results in the same type as promoting the numerator/denominator type with the float.

This small handful of promotion rules, together with the conversion methods discussed above, are sufficient to make rational numbers interoperate completely naturally with all of Julia's other numeric types – integers, floating-point numbers, and complex numbers. By providing appropriate conversion methods and promotion rules in the same manner, any user-defined numeric type can interoperate just as naturally with Julia's predefined numerics.

Chapter 18

Interfaces

A lot of the power and extensibility in Julia comes from a collection of informal interfaces. By extending a few specific methods to work for a custom type, objects of that type not only receive those functionalities, but they are also able to be used in other methods that are written to generically build upon those behaviors.

18.1 Iteration

Required methods		Brief description
		Returns the initial iteration state
		Returns the current item and the next state
		Tests if there are any items remaining
Important optional methods	**Default definition**	**Brief description**
		One of , , , or as appropriate
		Either or as appropriate
		The type the items returned by
	(*undefined*)	The number of items, if known
	(*undefined*)	The number of items in each dimension, if known

Value returned by	Required Methods
	and
	(*none*)
	(*none*)

Value returned by	Required Methods
	(*none*)

Sequential iteration is implemented by the methods , , and . Instead of mutating objects as they are iterated over, Julia provides these three methods to keep track of the iteration state externally from the object. The method returns the initial state for the iterable object . That state gets passed along to , which tests if there are any elements remaining, and , which returns a tuple containing the current element and an updated . The object can be anything, and is generally considered to be an implementation detail private to the iterable object.

Any object defines these three methods is iterable and can be used in the many functions that rely upon iteration. It can also be used directly in a loop since the syntax:

is translated into:

A simple example is an iterable sequence of square numbers with a defined length:

With only , , and definitions, the type is already pretty powerful. We can iterate over all the elements:

We can use many of the builtin methods that work with iterables, like , and :

There are a few more methods we can extend to give Julia more information about this iterable collection. We know that the elements in a sequence will always be . By extending the method, we can give that information to Julia and help it make more specialized code in the more complicated methods. We also know the number of elements in our sequence, so we can extend , too.

Now, when we ask Julia to all the elements into an array it can preallocate a of the right size instead of blindly ing each element into a :

While we can rely upon generic implementations, we can also extend specific methods where we know there is a simpler algorithm. For example, there's a formula to compute the sum of squares, so we can override the generic iterative version with a more performant solution:

This is a very common pattern throughout the Julia standard library: a small set of required methods define an informal interface that enable many fancier behaviors. In some cases, types will want to additionally specialize those extra behaviors when they know a more efficient algorithm can be used in their specific case.

18.2 Indexing

Methods to implement	Brief description
	, indexed element access
	, indexed assignment
	The last index, used in

For the iterable above, we can easily compute the th element of the sequence by squaring it. We can expose this as an indexing expression . To opt into this behavior, simply needs to define :

Additionally, to support the syntax , we must define to specify the last valid index:

Note, though, that the above *only* defines with one integer index. Indexing with anything other than an will throw a saying that there was no matching method. In order to support indexing with ranges or vectors of s, separate methods must be written:

While this is starting to support more of the indexing operations supported by some of the builtin types, there's still quite a number of behaviors missing. This sequence is starting to look more and more like a vector as we've added behaviors to it. Instead of defining all these behaviors ourselves, we can officially define it as a subtype of an .

18.3 Abstract Arrays

Methods to implement		Brief description
		Returns a tuple containing the dimensions of
		(if) Linear scalar indexing
		(if , where) N-dimensional scalar indexing
		(if) Scalar indexed assignment
		(if , where) N-dimensional scalar indexed assignment
Optional methods	**Default definition**	**Brief description**
		Returns either or . See the description below.
	defined in terms of scalar	Multidimensional and nonscalar indexing
	defined in terms of scalar	Multidimensional and nonscalar indexed assignment
//	defined in terms of scalar	Iteration
		Number of elements
		Return a mutable array with the same shape and element type
		Return a mutable array with the same shape and the specified element type
		Return a mutable array with the same element type and size *dims*
		Return a mutable array with the specified element type and size
Non-traditional indices	**Default definition**	**Brief description**
		Return the of valid indices
		Return a mutable array with the specified indices (see below)
		Return an array similar to with the specified indices (see below)

If a type is defined as a subtype of , it inherits a very large set of rich behaviors including iteration and multidimensional indexing built on top of single-element access. See the arrays manual page and standard library section for more supported methods.

A key part in defining an subtype is . Since indexing is such an important part of an array and often occurs in hot loops, it's important to make both indexing and indexed assignment as efficient as possible. Array data structures are typically defined in one of two ways: either it most efficiently accesses its elements using just one index (linear indexing) or it intrinsically accesses the elements with indices specified for every dimension. These two modalities are identified by Julia as and . Converting a linear index to multiple indexing subscripts is typically very expensive, so this provides a traits-based mechanism to enable efficient generic code for all array types.

This distinction determines which scalar indexing methods the type must define. arrays are simple: just define . When the array is subsequently indexed with a multidimensional set of indices, the fallback efficiently converts the indices into one linear index and then calls the above method. arrays, on the other hand, require methods to be defined for each supported dimensionality with indices. For example, the built-in type only supports two dimensions, so it just defines . The same holds for .

Returning to the sequence of squares from above, we could instead define it as a subtype of an :

Note that it's very important to specify the two parameters of the ; the first defines the , and the second defines the . That supertype and those three methods are all it takes for to be an iterable, indexable, and completely functional array:

As a more complicated example, let's define our own toy N-dimensional sparse-like array type built on top of :

Notice that this is an array, so we must manually define and at the dimensionality of the array. Unlike the , we are able to define , and so we can mutate the array:

The result of indexing an can itself be an array (for instance when indexing by a). The fallback methods use to allocate an of the appropriate size and element type, which is filled in using the basic indexing method described above. However, when implementing an array wrapper you often want the result to be wrapped as well:

In this example it is accomplished by defining to create the appropriate wrapped array. (Note that while supports 1- and 2-argument forms, in most case you only need to specialize the 3-argument form.) For this to work it's important that is mutable (supports). Defining , and for also makes it possible to the array:

In addition to all the iterable and indexable methods from above, these types can also interact with each other and use most of the methods defined in the standard library for :

If you are defining an array type that allows non-traditional indexing (indices that start at something other than 1), you should specialize . You should also specialize so that the argument (ordinarily a size-tuple) can accept objects, perhaps range-types of your own design. For more information, see Arrays with custom indices.

Chapter 19

Modules

Modules in Julia are separate variable workspaces, i.e. they introduce a new global scope. They are delimited syntactically, inside . Modules allow you to create top-level definitions (aka global variables) without worrying about name conflicts when your code is used together with somebody else's. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting).

The following example demonstrates the major features of modules. It is not meant to be run, but is shown for illustrative purposes:

Note that the style is not to indent the body of the module, since that would typically lead to whole files being indented.

This module defines a type , and two functions. Function and type are exported, and so will be available for importing into other modules. Function is private to .

The statement means that a module called will be available for resolving names as needed. When a global variable is encountered that has no definition in the current module, the system will search for it among variables exported by and import it if it is found there. This means that all uses of that global within the current module will resolve to the definition of that variable in .

The statement is a syntactic shortcut for .

The keyword supports all the same syntax as , but only operates on a single name at a time. It does not add modules to be searched the way does. also differs from in that functions must be imported using to be extended with new methods.

In above we wanted to add a method to the standard function, so we had to write . Functions whose names are only visible via cannot be extended.

The keyword explicitly imports all names exported by the specified module, as if were individually used on all of them.

Once a variable is made visible via or , a module may not create its own variable with the same name. Imported variables are read-only; assigning to a global variable always affects a variable owned by the current module, or else raises an error.

19.1 Summary of module usage

To load a module, two main keywords can be used: and . To understand their differences, consider the following example:

In this module we export the and functions (with the keyword), and also have the non-exported function . There are several different ways to load the Module and its inner functions into the current workspace:

Import Command	What is brought into scope	Available for method extension
	All ed names (and), , and	, and
	and	
	and	
	, and	, and
	and	and
	and	and
	All ed names (and)	and

Modules and files

Files and file names are mostly unrelated to modules; modules are associated only with module expressions. One can have multiple files per module, and multiple modules per file:

Including the same code in different modules provides mixin-like behavior. One could use this to run the same code with different base definitions, for example testing code by running it with ”safe” versions of some operators:

Standard modules

There are three important standard modules: Main, Core, and Base.

Main is the top-level module, and Julia starts with Main set as the current module. Variables defined at the prompt go in Main, and lists variables in Main.

Core contains all identifiers considered ”built in” to the language, i.e. part of the core language and not libraries. Every module implicitly specifies , since you can't do anything without those definitions.

Base is the standard library (the contents of base/). All modules implicitly contain , since this is needed in the vast majority of cases.

Default top-level definitions and bare modules

In addition to , modules also automatically contain a definition of the function, which evaluates expressions within the context of that module.

If these default definitions are not wanted, modules can be defined using the keyword instead (note: is still imported, as per above). In terms of , a standard looks like this:

Relative and absolute module paths

Given the statement , the system looks for within . If the module does not exist, the system attempts to , which typically results in loading code from an installed package.

However, some modules contain submodules, which means you sometimes need to access a module that is not directly available in . There are two ways to do this. The first is to use an absolute path, for example . The second is to use a relative path, which makes it easier to import submodules of the current module or any of its enclosing modules:

Here module contains a submodule , and code in wants the contents of to be visible. This is done by starting the path with a period. Adding more leading periods moves up additional levels in the module hierarchy. For example would look for in 's enclosing module rather than in itself.

Note that relative-import qualifiers are only valid in and statements.

Module file paths

The global variable contains the directories Julia searches for modules when calling . It can be extended using :

Putting this statement in the file will extend on every Julia startup. Alternatively, the module load path can be extended by defining the environment variable .

Namespace miscellanea

If a name is qualified (e.g.), then it can be accessed even if it is not exported. This is often useful when debugging. It can also have methods added to it by using the qualified name as the function name. However, due to syntactic ambiguities that arise, if you wish to add methods to a function in a different module whose name contains only symbols, such as an operator, for example, you must use to refer to it. If the operator is more than one character in length you must surround it in brackets, such as: .

Macro names are written with in import and export statements, e.g. . Macros in other modules can be invoked as or .

The syntax does not work to assign a global in another module; global assignment is always module-local.

A variable can be "reserved" for the current module without assigning to it by declaring it as at the top level. This can be used to prevent name conflicts for globals initialized after load time.

Module initialization and precompilation

Large modules can take several seconds to load because executing all of the statements in a module often involves compiling a large amount of code. Julia provides the ability to create precompiled versions of modules to reduce this time.

To create an incremental precompiled module file, add at the top of your module file (before the starts). This will cause it to be automatically compiled the first time it is imported. Alternatively, you can manually call . The resulting cache files will be stored in . Subsequently, the module is automatically recompiled upon whenever any of its dependencies change; dependencies are modules it imports, the Julia build, files it includes, or explicit dependencies declared by in the module file(s).

For file dependencies, a change is determined by examining whether the modification time (mtime) of each file loaded by or added explicitly by is unchanged, or equal to the modification time truncated to the nearest second (to accommodate systems that can't copy mtime with sub-second accuracy). It also takes into account whether the path to the file chosen by the search logic in matches the path that had created the precompile file.

It also takes into account the set of dependencies already loaded into the current process and won't recompile those modules, even if their files change or disappear, in order to avoid creating incompatibilities between the running system

and the precompile cache. If you want to have changes to the source reflected in the running system, you should call on the module you changed, and any module that depended on it in which you want to see the change reflected.

Precompiling a module also recursively precompiles any modules that are imported therein. If you know that it is *not* safe to precompile your module (for the reasons described below), you should put in the module file to cause to throw an error (and thereby prevent the module from being imported by any other precompiled module).

should *not* be used in a module unless all of its dependencies are also using . Failure to do so can result in a runtime error when loading the module.

In order to make your module work with precompilation, however, you may need to change your module to explicitly separate any initialization steps that must occur at *runtime* from steps that can occur at *compile time*. For this purpose, Julia allows you to define an function in your module that executes any initialization steps that must occur at runtime. This function will not be called during compilation (or). You may, of course, call it manually if necessary, but the default is to assume this function deals with computing state for the local machine, which does not need to be – or even should not be – captured in the compiled image. It will be called after the module is loaded into a process, including if it is being loaded into an incremental compile (), but not if it is being loaded into a full-compilation process.

In particular, if you define a in a module, then Julia will call immediately *after* the module is loaded (e.g., by , , or) at runtime for the *first* time (i.e., is only called once, and only after all statements in the module have been executed). Because it is called after the module is fully imported, any submodules or other imported modules have their functions called *before* the of the enclosing module.

Two typical uses of are calling runtime initialization functions of external C libraries and initializing global constants that involve pointers returned by external libraries. For example, suppose that we are calling a C library that requires us to call a initialization function at runtime. Suppose that we also want to define a global constant that holds the return value of a function defined by – this constant must be initialized at runtime (not at compile time) because the pointer address will change from run to run. You could accomplish this by defining the following function in your module:

Notice that it is perfectly possible to define a global inside a function like ; this is one of the advantages of using a dynamic language. But by making it a constant at global scope, we can ensure that the type is known to the compiler and allow it to generate better optimized code. Obviously, any other globals in your module that depends on would also have to be initialized in .

Constants involving most Julia objects that are not produced by do not need to be placed in : their definitions can be precompiled and loaded from the cached module image. This includes complicated heap-allocated objects like arrays. However, any routine that returns a raw pointer value must be called at runtime for precompilation to work (Ptr objects will turn into null pointers unless they are hidden inside an isbits object). This includes the return values of the Julia functions and .

Dictionary and set types, or in general anything that depends on the output of a method, are a trickier case. In the common case where the keys are numbers, strings, symbols, ranges, , or compositions of these types (via arrays, tuples, sets, pairs, etc.) they are safe to precompile. However, for a few other key types, such as or and generic user-defined types where you haven't defined a method, the fallback method depends on the memory address of the object (via its) and hence may change from run to run. If you have one of these key types, or if you aren't sure, to be safe you can initialize this dictionary from within your function. Alternatively, you can use the dictionary type, which is specially handled by precompilation so that it is safe to initialize at compile-time.

When using precompilation, it is important to keep a clear sense of the distinction between the compilation phase and the execution phase. In this mode, it will often be much more clearly apparent that Julia is a compiler which allows execution of arbitrary Julia code, not a standalone interpreter that also generates compiled code.

Other known potential failure scenarios include:

1. Global counters (for example, for attempting to uniquely identify objects) Consider the following code snippet:

   while the intent of this code was to give every instance a unique id, the counter value is recorded at the end of compilation. All subsequent usages of this incrementally compiled module will start from that same counter value.

   Note that (which works by hashing the memory pointer) has similar issues (see notes on usage below).

   One alternative is to use a macro to capture and store it alone with the current value, however, it may be better to redesign the code to not depend on this global state.

2. Associative collections (such as and) need to be re-hashed in . (In the future, a mechanism may be provided to register an initializer function.)

3. Depending on compile-time side-effects persisting through load-time. Example include: modifying arrays or other variables in other Julia modules; maintaining handles to open files or devices; storing pointers to other system resources (including memory);

4. Creating accidental "copies" of global state from another module, by referencing it directly instead of via its lookup path. For example, (in global scope):

Several additional restrictions are placed on the operations that can be done while precompiling code to help the user avoid other wrong-behavior situations:

1. Calling to cause a side-effect in another module. This will also cause a warning to be emitted when the incremental precompile flag is set.

2. statements from local scope after has been started (see issue #12010 for plans to add an error for this)

3. Replacing a module (or calling) is a runtime error while doing an incremental precompile.

A few other points to be aware of:

1. No code reload / cache invalidation is performed after changes are made to the source files themselves, (including by), and no cleanup is done after

2. The memory sharing behavior of a reshaped array is disregarded by precompilation (each view gets its own copy)

3. Expecting the filesystem to be unchanged between compile-time and runtime e.g. / to find resources at runtime, or the BinDeps macro. Sometimes this is unavoidable. However, when possible, it can be good practice to copy resources into the module at compile-time so they won't need to be found at runtime.

4. objects and finalizers are not currently handled properly by the serializer (this will be fixed in an upcoming release).

5. It is usually best to avoid capturing references to instances of internal metadata objects such as , , , , and fields of those objects, as this can confuse the serializer and may not lead to the outcome you desire. It is not necessarily an error to do this, but you simply need to be prepared that the system will try to copy some of these and to create a single unique instance of others.

It is sometimes helpful during module development to turn off incremental precompilation. The command line flag enables you to toggle module precompilation on and off. When Julia is started with the serialized modules in the compile cache are ignored when loading modules and module dependencies. can still be called manually and it will respect directives for the module. The state of this command line flag is passed to to disable automatic precompilation triggering when installing, updating, and explicitly building packages.

Chapter 20

Documentation

Julia enables package developers and users to document functions, types and other objects easily via a built-in documentation system since Julia 0.4.

The basic syntax is very simple: any string appearing at the top-level right before an object (function, macro, type or instance) will be interpreted as documenting it (these are called *docstrings*). Here is a very simple example:

Documentation is interpreted as Markdown, so you can use indentation and code fences to delimit code examples from text. Technically, any object can be associated with any other as metadata; Markdown happens to be the default, but one can construct other string macros and pass them to the macro just as well.

Here is a more complex example, still using Markdown:

As in the example above, we recommend following some simple conventions when writing documentation:

1. Always show the signature of a function at the top of the documentation, with a four-space indent so that it is printed as Julia code.

   This can be identical to the signature present in the Julia code (like), or a simplified form. Optional arguments should be represented with their default values (i.e.) when possible, following the actual Julia syntax. Optional arguments which do not have a default value should be put in brackets (i.e. and). An alternative solution is to use several lines: one without optional arguments, the other(s) with them. This solution can also be used to document several related methods of a given function. When a function accepts many keyword arguments, only include a placeholder in the signature (i.e.), and give the complete list under an section (see point 4 below).

2. Include a single one-line sentence describing what the function does or what the object represents after the simplified signature block. If needed, provide more details in a second paragraph, after a blank line.

   The one-line sentence should use the imperative form (”Do this”, ”Return that”) instead of the third person (do not write ”Returns the length...”) when documenting functions. It should end with a period. If the meaning of a function cannot be summarized easily, splitting it into separate composable parts could be beneficial (this should not be taken as an absolute requirement for every single case though).

3. Do not repeat yourself.

   Since the function name is given by the signature, there is no need to start the documentation with ”The function ...”: go straight to the point. Similarly, if the signature specifies the types of the arguments, mentioning them in the description is redundant.

4. Only provide an argument list when really necessary.

   For simple functions, it is often clearer to mention the role of the arguments directly in the description of the function’s purpose. An argument list would only repeat information already provided elsewhere. However, providing an argument list can be a good idea for complex functions with many arguments (in particular keyword arguments). In that case, insert it after the general description of the function, under an header, with one bullet for each argument. The list should mention the types and default values (if any) of the arguments:

5. Include any code examples in an section.

   Examples should, whenever possible, be written as *doctests*. A *doctest* is a fenced code block (see Code blocks) starting with and contains any number of prompts together with inputs and expected outputs that mimic the Julia REPL.

   For example in the following docstring a variable is defined and the expected result, as printed in a Julia REPL, appears afterwards:

   Warning

   Calling and other RNG-related functions should be avoided in doctests since they will not produce consistent outputs during different Julia sessions. If you would like to show some random number generation related functionality, one option is to explicitly construct and seed your own (or other pseudorandom number generator) and pass it to the functions you are doctesting.

Operating system word size (or) as well as path separator differences (or) will also affect the reproducibility of some doctests.

Note that whitespace in your doctest is significant! The doctest will fail if you misalign the output of pretty-printing an array, for example.

You can then run to run all the doctests in the Julia Manual, which will ensure that your example works.

Examples that are untestable should be written within fenced code blocks starting with so that they are highlighted correctly in the generated documentation.

Tip

Wherever possible examples should be **self-contained** and **runnable** so that readers are able to try them out without having to include any dependencies.

6. Use backticks to identify code and equations.

   Julia identifiers and code excerpts should always appear between backticks to enable highlighting. Equations in the LaTeX syntax can be inserted between double backticks . Use Unicode characters rather than their LaTeX escape sequence, i.e. rather than .

7. Place the starting and ending characters on lines by themselves.

   That is, write:

   rather than:

   This makes it more clear where docstrings start and end.

8. Respect the line length limit used in the surrounding code.

   Docstrings are edited using the same tools as code. Therefore, the same conventions should apply. It it advised to add line breaks after 92 characters.

20.1 Accessing Documentation

Documentation can be accessed at the REPL or in IJulia by typing followed by the name of a function or macro, and pressing . For example,

will bring up docs for the relevant function, macro or string macro respectively. In Juno using will bring up documentation for the object under the cursor.

20.2 Functions & Methods

Functions in Julia may have multiple implementations, known as methods. While it's good practice for generic functions to have a single purpose, Julia allows methods to be documented individually if necessary. In general, only the most generic method should be documented, or even the function itself (i.e. the object created without any methods by). Specific methods should only be documented if their behaviour differs from the more generic ones. In any case, they should not repeat the information provided elsewhere. For example:

When retrieving documentation for a generic function, the metadata for each method is concatenated with the function, which can of course be overridden for custom types.

20.3 Advanced Usage

The macro associates its first argument with its second in a per-module dictionary called . By default, documentation is expected to be written in Markdown, and the string macro simply creates an object representing the Markdown content. In the future it is likely to do more advanced things such as allowing for relative image or link paths.

When used for retrieving documentation, the macro (or equally, the function) will search all dictionaries for metadata relevant to the given object and return it. The returned object (some Markdown content, for example) will by default display itself intelligently. This design also makes it easy to use the doc system in a programmatic way; for example, to re-use documentation between different versions of a function:

Or for use with Julia's metaprogramming functionality:

Documentation written in non-toplevel blocks, such as , , , and , is added to the documentation system as blocks are evaluated. For example:

will add documentation to when the condition is . Note that even if goes out of scope at the end of the block, its documentation will remain.

Dynamic documentation

Sometimes the appropriate documentation for an instance of a type depends on the field values of that instance, rather than just on the type itself. In these cases, you can add a method to for your custom type that returns the documentation on a per-instance basis. For instance,

 will display "Documentation for MyType with value x" while will display "Documentation for MyType with value y".

20.4 Syntax Guide

A comprehensive overview of all documentable Julia syntax.

In the following examples is used to illustrate an arbitrary docstring which may be one of the follow four variants and contain arbitrary text:

should only be used when the docstring contains or characters that should not be parsed by Julia such as LaTeX syntax or Julia source code examples containing interpolation.

Functions and Methods

Adds docstring to . The first version is the preferred syntax, however both are equivalent.

Adds docstring to .

Adds docstring to two s, namely and .

Macros

Adds docstring to the macro definition.

Adds docstring to the macro named .

Types

Adds the docstring to types , , and .

Adds docstring to type , to field and to field . Also applicable to types.

Modules

Adds docstring to the . Adding the docstring above the is the preferred syntax, however both are equivalent.

Documenting a by placing a docstring above the expression automatically imports into the module. These imports must be done manually when the module expression is not documented. Empty s cannot be documented.

Global Variables

Adds docstring to the s , , and .

s are used to store a reference to a particular in a without storing the referenced value itself.

> **Note**
>
> When a definition is only used to define an alias of another definition, such as is the case with the function and its alias in , do not document the alias and instead document the actual function.
>
> If the alias is documented and not the real definition then the docsystem (mode) will not return the docstring attached to the alias when the real definition is searched for.
>
> For example you should write
>
> rather than

Adds docstring to the value associated with . Users should prefer documenting at it's definition.

Multiple Objects

Adds docstring to and each of which should be a documentable expression. This syntax is equivalent to

Any number of expressions many be documented together in this way. This syntax can be useful when two functions are related, such as non-mutating and mutating versions and .

Macro-generated code

Adds docstring to expression generated by expanding . This allows for expressions decorated with , , , or any other macro to be documented in the same way as undecorated expressions.

Macro authors should take note that only macros that generate a single expression will automatically support docstrings. If a macro returns a block containing multiple subexpressions then the subexpression that should be documented must be marked using the macro.

The macro makes use of to allow for documenting s. Examining it's definition should serve as an example of how to use correctly.

– Macro.

Low-level macro used to mark expressions returned by a macro that should be documented. If more than one expression is marked then the same docstring is applied to each expression.

has no effect when a macro that uses it is not documented.

20.5 Markdown syntax

The following markdown syntax is supported in Julia.

Inline elements

Here "inline" refers to elements that can be found within blocks of text, i.e. paragraphs. These include the following elements.

Bold

Surround words with two asterisks, , to display the enclosed text in boldface.

Italics

Surround words with one asterisk, , to display the enclosed text in italics.

Literals

Surround text that should be displayed exactly as written with single backticks, .

Literals should be used when writing text that refers to names of variables, functions, or other parts of a Julia program.

> **Tip**
>
> To include a backtick character within literal text use three backticks rather than one to enclose the text.
>
> By extension any odd number of backticks may be used to enclose a lesser number of backticks.

LaTeX

Surround text that should be displayed as mathematics using LaTeXsyntax with double backticks, .

> **Tip**
>
> As with literals in the previous section, if literal backticks need to be written within double backticks use an even number greater than two. Note that if a single literal backtick needs to be included within LaTeXmarkup then two enclosing backticks is sufficient.

Links

Links to either external or internal addresses can be written using the following syntax, where the text enclosed in square brackets, , is the name of the link and the text enclosed in parentheses, , is the URL.

It's also possible to add cross-references to other documented functions/methods/variables within the Julia documentation itself. For example:

This will create a link in the generated docs to the documentation (which has more information about what this function actually does). It's good to include cross references to mutating/non-mutating versions of a function, or to highlight a difference between two similar-seeming functions.

> **Note**
>
> The above cross referencing is *not* a Markdown feature, and relies on Documenter.jl, which is used to build base Julia's documentation.

Footnote references

Named and numbered footnote references can be written using the following syntax. A footnote name must be a single alphanumeric word containing no punctuation.

> **Note**
>
> The text associated with a footnote can be written anywhere within the same page as the footnote reference. The syntax used to define the footnote text is discussed in the Footnotes section below.

Toplevel elements

The following elements can be written either at the ”toplevel” of a document or within another ”toplevel” element.

Paragraphs

A paragraph is a block of plain text, possibly containing any number of inline elements defined in the Inline elements section above, with one or more blank lines above and below it.

Headers

A document can be split up into different sections using headers. Headers use the following syntax:

A header line can contain any inline syntax in the same way as a paragraph can.

> **Tip**
>
> Try to avoid using too many levels of header within a single document. A heavily nested document may be indicative of a need to restructure it or split it into several pages covering separate topics.

Code blocks

Source code can be displayed as a literal block using an indent of four spaces as shown in the following example.

Additionally, code blocks can be enclosed using triple backticks with an optional ”language” to specify how a block of code should be highlighted.

> **Note**
>
> ”Fenced” code blocks, as shown in the last example, should be prefered over indented code blocks since there is no way to specify what language an indented code block is written in.

Block quotes

Text from external sources, such as quotations from books or websites, can be quoted using characters prepended to each line of the quote as follows.

Note that a single space must appear after the character on each line. Quoted blocks may themselves contain other toplevel or inline elements.

Images

The syntax for images is similar to the link syntax mentioned above. Prepending a character to a link will display an image from the specified URL rather than a link to it.

Lists

Unordered lists can be written by prepending each item in a list with either , , or .

Note the two spaces before each and the single space after each one.

Lists can contain other nested toplevel elements such as lists, code blocks, or quoteblocks. A blank line should be left between each list item when including any toplevel elements within a list.

Note

The contents of each item in the list must line up with the first line of the item. In the above example the fenced code block must be indented by four spaces to align with the in .

Ordered lists are written by replacing the ”bullet” character, either , , or , with a positive integer followed by either or .

An ordered list may start from a number other than one, as in the second list of the above example, where it is numbered from five. As with unordered lists, ordered lists can contain nested toplevel elements.

Display equations

Large LaTeXequations that do not fit inline within a paragraph may be written as display equations using a fenced code block with the ”language” as in the example below.

Footnotes

This syntax is paired with the inline syntax for Footnote references. Make sure to read that section as well.

Footnote text is defined using the following syntax, which is similar to footnote reference syntax, aside from the character that is appended to the footnote label.

Note

No checks are done during parsing to make sure that all footnote references have matching footnotes.

Horizontal rules

The equivalent of an HTML tag can be written using the following syntax:

Tables

Basic tables can be written using the syntax described below. Note that markdown tables have limited features and cannot contain nested toplevel elements unlike other elements discussed above – only inline elements are allowed. Tables must always contain a header row with column names. Cells cannot span multiple rows or columns of the table.

Note

As illustrated in the above example each column of characters must be aligned vertically.

A character on either end of a column's header separator (the row containing characters) specifies whether the row is left-aligned, right-aligned, or (when appears on both ends) center-aligned. Providing no characters will default to right-aligning the column.

Admonitions

Specially formatted blocks with titles such as "Notes", "Warning", or "Tips" are known as admonitions and are used when some part of a document needs special attention. They can be defined using the following syntax:

Admonitions, like most other toplevel elements, can contain other toplevel elements. When no title text, specified after the admonition type in double quotes, is included then the title used will be the type of the block, i.e. in the case of the admonition.

20.6 Markdown Syntax Extensions

Julia's markdown supports interpolation in a very similar way to basic string literals, with the difference that it will store the object itself in the Markdown tree (as opposed to converting it to a string). When the Markdown content is rendered the usual methods will be called, and these can be overridden as usual. This design allows the Markdown to be extended with arbitrarily complex features (such as references) without cluttering the basic syntax.

In principle, the Markdown parser itself can also be arbitrarily extended by packages, or an entirely custom flavour of Markdown can be used, but this should generally be unnecessary.

Chapter 21

Metaprogramming

The strongest legacy of Lisp in the Julia language is its metaprogramming support. Like Lisp, Julia represents its own code as a data structure of the language itself. Since code is represented by objects that can be created and manipulated from within the language, it is possible for a program to transform and generate its own code. This allows sophisticated code generation without extra build steps, and also allows true Lisp-style macros operating at the level of abstract syntax trees. In contrast, preprocessor "macro" systems, like that of C and C++, perform textual manipulation and substitution before any actual parsing or interpretation occurs. Because all data types and code in Julia are represented by Julia data structures, powerful reflection capabilities are available to explore the internals of a program and its types just like any other data.

21.1 Program representation

Every Julia program starts life as a string:

What happens next?

The next step is to parse each string into an object called an expression, represented by the Julia type :

objects contain two parts:

- a identifying the kind of expression. A symbol is an interned string identifier (more discussion below).

- the expression arguments, which may be symbols, other expressions, or literal values:

Expressions may also be constructed directly in prefix notation:

The two expressions constructed above – by parsing and by direct construction – are equivalent:

The key point here is that Julia code is internally represented as a data structure that is accessible from the language itself.

The function provides indented and annotated display of objects:

objects may also be nested:

Another way to view expressions is with Meta.show_sexpr, which displays the S-expression form of a given , which may look very familiar to users of Lisp. Here's an example illustrating the display on a nested :

Symbols

The character has two syntactic purposes in Julia. The first form creates a , an interned string used as one building-block of expressions:

The constructor takes any number of arguments and creates a new symbol by concatenating their string representations together:

In the context of an expression, symbols are used to indicate access to variables; when an expression is evaluated, a symbol is replaced with the value bound to that symbol in the appropriate scope.

Sometimes extra parentheses around the argument to are needed to avoid ambiguity in parsing.:

21.2 Expressions and evaluation

Quoting

The second syntactic purpose of the character is to create expression objects without using the explicit constructor. This is referred to as *quoting*. The character, followed by paired parentheses around a single statement of Julia code, produces an object based on the enclosed code. Here is example of the short form used to quote an arithmetic expression:

(to view the structure of this expression, try and , or use as above or)

Note that equivalent expressions may be constructed using or the direct form:

Expressions provided by the parser generally only have symbols, other expressions, and literal values as their args, whereas expressions constructed by Julia code can have arbitrary run-time values without literal forms as args. In this specific example, and are symbols, is a subexpression, and is a literal 64-bit signed integer.

There is a second syntactic form of quoting for multiple expressions: blocks of code enclosed in . Note that this form introduces elements to the expression tree, which must be considered when directly manipulating an expression tree generated from blocks. For other purposes, and blocks are treated identically.

Interpolation

Direct construction of objects with value arguments is powerful, but constructors can be tedious compared to "normal" Julia syntax. As an alternative, Julia allows "splicing" or interpolation of literals or expressions into quoted expressions. Interpolation is indicated by the prefix.

In this example, the literal value of is interpolated:

Interpolating into an unquoted expression is not supported and will cause a compile-time error:

In this example, the tuple is interpolated as an expression into a conditional test:

Interpolating symbols into a nested expression requires enclosing each symbol in an enclosing quote block:

The use of for expression interpolation is intentionally reminiscent of string interpolation and command interpolation. Expression interpolation allows convenient, readable programmatic construction of complex Julia expressions.

and effects

Given an expression object, one can cause Julia to evaluate (execute) it at global scope using :

Every module has its own function that evaluates expressions in its global scope. Expressions passed to are not limited to returning values – they can also have side-effects that alter the state of the enclosing module's environment:

Here, the evaluation of an expression object causes a value to be assigned to the global variable .

Since expressions are just objects which can be constructed programmatically and then evaluated, it is possible to dynamically generate arbitrary code which can then be run using . Here is a simple example:

The value of is used to construct the expression which applies the function to the value 1 and the variable . Note the important distinction between the way and are used:

- The value of the *variable* at expression construction time is used as an immediate value in the expression. Thus, the value of when the expression is evaluated no longer matters: the value in the expression is already , independent of whatever the value of might be.

- On the other hand, the *symbol* is used in the expression construction, so the value of the variable at that time is irrelevant – is just a symbol and the variable need not even be defined. At expression evaluation time, however, the value of the symbol is resolved by looking up the value of the variable .

Functions on essions

As hinted above, one extremely useful feature of Julia is the capability to generate and manipulate Julia code within Julia itself. We have already seen one example of a function returning objects: the function, which takes a string of Julia code and returns the corresponding . A function can also take one or more objects as arguments, and return another . Here is a simple, motivating example:

As another example, here is a function that doubles any numeric argument, but leaves expressions alone:

21.3 Macros

Macros provide a method to include generated code in the final body of a program. A macro maps a tuple of arguments to a returned *expression*, and the resulting expression is compiled directly rather than requiring a runtime call. Macro arguments may include expressions, literal values, and symbols.

Basics

Here is an extraordinarily simple macro:

Macros have a dedicated character in Julia's syntax: the (at-sign), followed by the unique name declared in a block. In this example, the compiler will replace all instances of with:

When is entered in the REPL, the expression executes immediately, thus we only see the evaluation result:

Now, consider a slightly more complex macro:

This macro takes one argument: . When is encountered, the quoted expression is *expanded* to interpolate the value of the argument into the final expression:

We can view the quoted return expression using the function (**important note:** this is an extremely useful tool for debugging macros):

We can see that the literal has been interpolated into the expression.

There also exists a macro that is perhaps a bit more convenient than the function:

Hold up: why macros?

We have already seen a function in a previous section. In fact, is also such a function. So, why do macros exist?

Macros are necessary because they execute when code is parsed, therefore, macros allow the programmer to generate and include fragments of customized code *before* the full program is run. To illustrate the difference, consider the following example:

The first call to is executed when is called. The resulting expression contains *only* the second :

Macro invocation

Macros are invoked with the following general syntax:

Note the distinguishing before the macro name and the lack of commas between the argument expressions in the first form, and the lack of whitespace after in the second form. The two styles should not be mixed. For example, the following syntax is different from the examples above; it passes the tuple as one argument to the macro:

It is important to emphasize that macros receive their arguments as expressions, literals, or symbols. One way to explore macro arguments is to call the function within the macro body:

In addition to the given argument list, every macro is passed extra arguments named and .

The argument provides information (in the form of a object) about the parser location of the sign from the macro invocation. This allows macros to include better error diagnostic information, and is commonly used by logging, string-parser macros, and docs, for example, as well as to implement the , , and macros.

The location information can be accessed by referencing and :

The argument provides information (in the form of a object) about the expansion context of the macro invocation. This allows macros to look up contextual information, such as existing bindings, or to insert the value as an extra argument to a runtime function call doing self-reflection in the current module.

Building an advanced macro

Here is a simplified definition of Julia's macro:

This macro can be used like this:

In place of the written syntax, the macro call is expanded at parse time to its returned result. This is equivalent to writing:

That is, in the first call, the expression is spliced into the test condition slot, while the value of is spliced into the assertion message slot. The entire expression, thus constructed, is placed into the syntax tree where the macro call occurs. Then at execution time, if the test expression evaluates to true, then is returned, whereas if the test is false, an error is raised indicating the asserted expression that was false. Notice that it would not be possible to write this as a function, since

only the *value* of the condition is available and it would be impossible to display the expression that computed it in the error message.

The actual definition of in the standard library is more complicated. It allows the user to optionally specify their own error message, instead of just printing the failed expression. Just like in functions with a variable number of arguments, this is specified with an ellipses following the last argument:

Now has two modes of operation, depending upon the number of arguments it receives! If there's only one argument, the tuple of expressions captured by will be empty and it will behave the same as the simpler definition above. But now if the user specifies a second argument, it is printed in the message body instead of the failing expression. You can inspect the result of a macro expansion with the aptly named macro:

There is yet another case that the actual macro handles: what if, in addition to printing "a should equal b," we wanted to print their values? One might naively try to use string interpolation in the custom message, e.g., , but this won't work as expected with the above macro. Can you see why? Recall from string interpolation that an interpolated string is rewritten to a call to . Compare:

So now instead of getting a plain string in , the macro is receiving a full expression that will need to be evaluated in order to display as expected. This can be spliced directly into the returned expression as an argument to the call; see for the complete implementation.

The macro makes great use of splicing into quoted expressions to simplify the manipulation of expressions inside the macro body.

Hygiene

An issue that arises in more complex macros is that of hygiene. In short, macros must ensure that the variables they introduce in their returned expressions do not accidentally clash with existing variables in the surrounding code they expand into. Conversely, the expressions that are passed into a macro as arguments are often *expected* to evaluate in the context of the surrounding code, interacting with and modifying the existing variables. Another concern arises from the fact that a macro may be called in a different module from where it was defined. In this case we need to ensure that all global variables are resolved to the correct module. Julia already has a major advantage over languages with textual macro expansion (like C) in that it only needs to consider the returned expression. All the other variables (such as in above) follow the normal scoping block behavior.

To demonstrate these issues, let us consider writing a macro that takes an expression as its argument, records the time, evaluates the expression, records the time again, prints the difference between the before and after times, and then has the value of the expression as its final value. The macro might look like this:

Here, we want , , and to be private temporary variables, and we want to refer to the function in the standard library, not to any variable the user might have (the same applies to). Imagine the problems that could occur if the user expression also contained assignments to a variable called , or defined its own variable. We might get errors, or mysteriously incorrect behavior.

Julia's macro expander solves these problems in the following way. First, variables within a macro result are classified as either local or global. A variable is considered local if it is assigned to (and not declared global), declared local, or used as a function argument name. Otherwise, it is considered global. Local variables are then renamed to be unique (using the function, which generates new symbols), and global variables are resolved within the macro definition environment. Therefore both of the above concerns are handled; the macro's locals will not conflict with any user variables, and and will refer to the standard library definitions.

One problem remains however. Consider the following use of this macro:

Here the user expression is a call to , but not the same function that the macro uses. It clearly refers to . Therefore we must arrange for the code in to be resolved in the macro call environment. This is done by ”escaping” the expression with :

An expression wrapped in this manner is left alone by the macro expander and simply pasted into the output verbatim. Therefore it will be resolved in the macro call environment.

This escaping mechanism can be used to ”violate” hygiene when necessary, in order to introduce or manipulate user variables. For example, the following macro sets to zero in the call environment:

This kind of manipulation of variables should be used judiciously, but is occasionally quite handy.

Getting the hygiene rules correct can be a formidable challenge. Before using a macro, you might want to consider whether a function closure would be sufficient. Another useful strategy is to defer as much work as possible to runtime. For example, many macros simply wrap their arguments in a QuoteNode or other similar Expr. Some examples of this include which simply returns , and , which simply returns .

To demonstrate, we might rewrite the example above as:

However, we don’t do this for a good reason: wrapping the in a new scope block (the anonymous function) also slightly changes the meaning of the expression (the scope of any variables in it), while we want to be usable with minimum impact on the wrapped code.

21.4 Code Generation

When a significant amount of repetitive boilerplate code is required, it is common to generate it programmatically to avoid redundancy. In most languages, this requires an extra build step, and a separate program to generate the repetitive code. In Julia, expression interpolation and allow such code generation to take place in the normal course of program execution. For example, the following code defines a series of operators on three arguments in terms of their 2-argument forms:

In this manner, Julia acts as its own preprocessor, and allows code generation from inside the language. The above code could be written slightly more tersely using the prefix quoting form:

This sort of in-language code generation, however, using the pattern, is common enough that Julia comes with a macro to abbreviate this pattern:

The macro rewrites this call to be precisely equivalent to the above longer versions. For longer blocks of generated code, the expression argument given to can be a block:

21.5 Non-Standard String Literals

Recall from Strings that string literals prefixed by an identifier are called non-standard string literals, and can have different semantics than un-prefixed string literals. For example:

- produces a regular expression object rather than a string
- is a byte array literal for .

Perhaps surprisingly, these behaviors are not hard-coded into the Julia parser or compiler. Instead, they are custom behaviors provided by a general mechanism that anyone can use: prefixed string literals are parsed as calls to specially-named macros. For example, the regular expression macro is just the following:

That's all. This macro says that the literal contents of the string literal should be passed to the macro and the result of that expansion should be placed in the syntax tree where the string literal occurs. In other words, the expression is equivalent to placing the following object directly into the syntax tree:

Not only is the string literal form shorter and far more convenient, but it is also more efficient: since the regular expression is compiled and the object is actually created *when the code is compiled*, the compilation occurs only once, rather than every time the code is executed. Consider if the regular expression occurs in a loop:

Since the regular expression is compiled and inserted into the syntax tree when this code is parsed, the expression is only compiled once instead of each time the loop is executed. In order to accomplish this without macros, one would have to write this loop like this:

Moreover, if the compiler could not determine that the regex object was constant over all loops, certain optimizations might not be possible, making this version still less efficient than the more convenient literal form above. Of course, there are still situations where the non-literal form is more convenient: if one needs to interpolate a variable into the regular expression, one must take this more verbose approach; in cases where the regular expression pattern itself is dynamic, potentially changing upon each loop iteration, a new regular expression object must be constructed on each iteration. In the vast majority of use cases, however, regular expressions are not constructed based on run-time data. In this majority of cases, the ability to write regular expressions as compile-time values is invaluable.

Like non-standard string literals, non-standard command literals exist using a prefixed variant of the command literal syntax. The command literal is parsed as . Julia itself does not contain any non-standard command literals, but packages can make use of this syntax. Aside from the different syntax and the suffix instead of the suffix, non-standard command literals behave exactly like non-standard string literals.

In the event that two modules provide non-standard string or command literals with the same name, it is possible to qualify the string or command literal with a module name. For instance, if both and provide non-standard string literal , then one can write or to disambiguate between the two.

The mechanism for user-defined string literals is deeply, profoundly powerful. Not only are Julia's non-standard literals implemented using it, but also the command literal syntax () is implemented with the following innocuous-looking macro:

Of course, a large amount of complexity is hidden in the functions used in this macro definition, but they are just functions, written entirely in Julia. You can read their source and see precisely what they do – and all they do is construct expression objects to be inserted into your program's syntax tree.

21.6 Generated functions

A very special macro is , which allows you to define so-called *generated functions*. These have the capability to generate specialized code depending on the types of their arguments with more flexibility and/or less code than what can be achieved with multiple dispatch. While macros work with expressions at parsing-time and cannot access the types of their inputs, a generated function gets expanded at a time when the types of the arguments are known, but the function is not yet compiled.

Instead of performing some calculation or action, a generated function declaration returns a quoted expression which then forms the body for the method corresponding to the types of the arguments. When called, the body expression is first evaluated and compiled, then the returned expression is compiled and run. To make this efficient, the result is often cached. And to make this inferable, only a limited subset of the language is usable. Thus, generated functions provide a flexible framework to move work from run-time to compile-time, at the expense of greater restrictions on the allowable constructs.

When defining generated functions, there are four main differences to ordinary functions:

1. You annotate the function declaration with the macro. This adds some information to the AST that lets the compiler know that this is a generated function.

2. In the body of the generated function you only have access to the *types* of the arguments – not their values – and any function that was defined *before* the definition of the generated function.

3. Instead of calculating something or performing some action, you return a *quoted expression* which, when evaluated, does what you want.

4. Generated functions must not *mutate* or *observe* any non-constant global state (including, for example, IO, locks, non-local dictionaries, or using). This means they can only read global constants, and cannot have any side effects. In other words, they must be completely pure. Due to an implementation limitation, this also means that they currently cannot define a closure or untyped generator.

It's easiest to illustrate this with an example. We can declare a generated function as

Note that the body returns a quoted expression, namely , rather than just the value of .

From the caller's perspective, they are very similar to regular functions; in fact, you don't have to know if you're calling a regular or generated function - the syntax and result of the call is just the same. Let's see how behaves:

So, we see that in the body of the generated function, is the *type* of the passed argument, and the value returned by the generated function, is the result of evaluating the quoted expression we returned from the definition, now with the *value* of .

What happens if we evaluate again with a type that we have already used?

Note that there is no printout of . We can see that the body of the generated function was only executed once here, for the specific set of argument types, and the result was cached. After that, for this example, the expression returned from the generated function on the first invocation was re-used as the method body. However, the actual caching behavior is an implementation-defined performance optimization, so it is invalid to depend too closely on this behavior.

The number of times a generated function is generated *might* be only once, but it *might* also be more often, or appear to not happen at all. As a consequence, you should *never* write a generated function with side effects - when, and how often, the side effects occur is undefined. (This is true for macros too - and just like for macros, the use of in a generated function is a sign that you're doing something the wrong way.) However, unlike macros, the runtime system cannot correctly handle a call to , so it is disallowed.

It is also important to see how functions interact with method redefinition. Following the principle that a correct function must not observe any mutable state or cause any mutation of global state, we see the following behavior. Observe that the generated function *cannot* call any method that was not defined prior to the *definition* of the generated function itself.

Initially has one definition

Define other operations that use :

We now add some new definitions for :

and compare how these results differ:

Each method of a generated function has its own view of defined functions:

The example generated function above did not do anything a normal function could not do (except printing the type on the first invocation, and incurring higher overhead). However, the power of a generated function lies in its ability to compute different quoted expressions depending on the types passed to it:

(although of course this contrived example would be more easily implemented using multiple dispatch...)

Abusing this will corrupt the runtime system and cause undefined behavior:

Since the body of the generated function is non-deterministic, its behavior, *and the behavior of all subsequent code* is undefined.

Don't copy these examples!

These examples are hopefully helpful to illustrate how generated functions work, both in the definition end and at the call site; however, *don't copy them*, for the following reasons:

- the function has side-effects (the call to), and it is undefined exactly when, how often or how many times these side-effects will occur
- the function solves a problem that is better solved with multiple dispatch - defining and will do the same thing, but it is both simpler and faster.
- the function is pathologically insane

Note that the set of operations that should not be attempted in a generated function is unbounded, and the runtime system can currently only detect a subset of the invalid operations. There are many other operations that will simply corrupt the runtime system without notification, usually in subtle ways not obviously connected to the bad definition. Because the function generator is run during inference, it must respect all of the limitations of that code.

Some operations that should not be attempted include:

1. Caching of native pointers.
2. Interacting with the contents or methods of Core.Inference in any way.
3. Observing any mutable state.
    - Inference on the generated function may be run at *any* time, including while your code is attempting to observe or mutate this state.
4. Taking any locks: C code you call out to may use locks internally, (for example, it is not problematic to call , even though most implementations require locks internally) but don't attempt to hold or acquire any while executing Julia code.
5. Calling any function that is defined after the body of the generated function. This condition is relaxed for incrementally-loaded precompiled modules to allow calling any function in the module.

Alright, now that we have a better understanding of how generated functions work, let's use them to build some more advanced (and valid) functionality...

An advanced example

Julia's base library has a function to calculate a linear index into an n-dimensional array, based on a set of n multilinear indices - in other words, to calculate the index that can be used to index into an array using , instead of . One possible implementation is the following:

The same thing can be done using recursion:

Both these implementations, although different, do essentially the same thing: a runtime loop over the dimensions of the array, collecting the offset in each dimension into the final index.

However, all the information we need for the loop is embedded in the type information of the arguments. Thus, we can utilize generated functions to move the iteration to compile-time; in compiler parlance, we use generated functions to manually unroll the loop. The body becomes almost identical, but instead of calculating the linear index, we build up an *expression* that calculates the index:

What code will this generate?

An easy way to find out is to extract the body into another (regular) function:

We can now execute and examine the expression it returns:

So, the method body that will be used here doesn't include a loop at all - just indexing into the two tuples, multiplication and addition/subtraction. All the looping is performed compile-time, and we avoid looping during execution entirely. Thus, we only loop *once per type*, in this case once per (except in edge cases where the function is generated more than once - see disclaimer above).

Chapter 22

Multi-dimensional Arrays

Julia, like most technical computing languages, provides a first-class array implementation. Most technical computing languages pay a lot of attention to their array implementation at the expense of other containers. Julia does not treat arrays in any special way. The array library is implemented almost completely in Julia itself, and derives its performance from the compiler, just like any other code written in Julia. As such, it's also possible to define custom array types by inheriting from See the manual section on the AbstractArray interface for more details on implementing a custom array type.

An array is a collection of objects stored in a multi-dimensional grid. In the most general case, an array may contain objects of type . For most computational purposes, arrays should contain objects of a more specific type, such as or .

In general, unlike many other technical computing languages, Julia does not expect programs to be written in a vectorized style for performance. Julia's compiler uses type inference and generates optimized code for scalar array indexing, allowing programs to be written in a style that is convenient and readable, without sacrificing performance, and using less memory at times.

In Julia, all arguments to functions are passed by reference. Some technical computing languages pass arrays by value, and this is convenient in many cases. In Julia, modifications made to input arrays within a function will be visible in the parent function. The entire Julia array library ensures that inputs are not modified by library functions. User code, if it needs to exhibit similar behavior, should take care to create a copy of inputs that it may modify.

22.1 Arrays

Basic Functions

Function	Description
	the type of the elements contained in
	the number of elements in
	the number of dimensions of
	a tuple containing the dimensions of
	the size of along dimension
	a tuple containing the valid indices of
	a range expressing the valid indices along dimension
	an efficient iterator for visiting each position in
	the stride (linear index distance between adjacent elements) along dimension
	a tuple of the strides in each dimension

Construction and Initialization

Many functions for constructing and initializing arrays are provided. In the following list of such functions, calls with a argument can either take a single tuple of dimension sizes or a series of dimension sizes passed as a variable number of arguments. Most of these functions also accept a first input , which is the element type of the array. If the type is omitted it will default to .

Function	Description
	an uninitialized dense
	an of all zeros
	an array of all zeros with the same type, element type and shape as
	an of all ones
	an array of all ones with the same type, element type and shape as
	a with all values
	a with all values and the same shape as
	a with all values
	a with all values and the same shape as
	an array containing the same data as , but with different dimensions
	copy
	copy , recursively copying its elements
	an uninitialized array of the same type as (dense, sparse, etc.), but with the specified element type and dimensions. The second and third arguments are both optional, defaulting to the element type and dimensions of if omitted.
	an array with the same binary data as , but with element type
	an with random, iid [1] and uniformly distributed values in the half-open interval $[0, 1)$
	an with random, iid and standard normally distributed values
	-by- identity matrix
	-by- identity matrix
	range of linearly spaced elements from to
	fill the array with the value
	an filled with the value

The syntax constructs a 1-d array (vector) of its arguments. If all arguments have a common promotion type then they get converted to that type using .

Concatenation

Arrays can be constructed and also concatenated using the following functions:

Function	Description
	concatenate input n-d arrays along the dimension
	shorthand for
	shorthand for

Scalar values passed to these functions are treated as 1-element arrays.

The concatenation functions are used so often that they have special syntax:

concatenates in both dimension 1 (with semicolons) and dimension 2 (with spaces).

[1] *iid*, independently and identically distributed.

Expression	Calls

Typed array initializers

An array with a specific element type can be constructed using the syntax . This will construct a 1-d array with element type , initialized to contain elements , , , etc. For example constructs a heterogeneous array that can contain any values.

Concatenation syntax can similarly be prefixed with a type to specify the element type of the result.

Comprehensions

Comprehensions provide a general and powerful way to construct arrays. Comprehension syntax is similar to set construction notation in mathematics:

The meaning of this form is that is evaluated with the variables , , etc. taking on each value in their given list of values. Values can be specified as any iterable object, but will commonly be ranges like or , or explicit arrays of values like . The result is an N-d dense array with dimensions that are the concatenation of the dimensions of the variable ranges , , etc. and each evaluation returns a scalar.

The following example computes a weighted average of the current element and its left and right neighbor along a 1-d grid. :

The resulting array type depends on the types of the computed elements. In order to control the type explicitly, a type can be prepended to the comprehension. For example, we could have requested the result in single precision by writing:

Generator Expressions

Comprehensions can also be written without the enclosing square brackets, producing an object known as a generator. This object can be iterated to produce values on demand, instead of allocating an array and storing them in advance (see Iteration). For example, the following expression sums a series without allocating memory:

When writing a generator expression with multiple dimensions inside an argument list, parentheses are needed to separate the generator from subsequent arguments:

All comma-separated expressions after are interpreted as ranges. Adding parentheses lets us add a third argument to :

Ranges in generators and comprehensions can depend on previous ranges by writing multiple keywords:

In such cases, the result is always 1-d.

Generated values can be filtered using the keyword:

Indexing

The general syntax for indexing into an n-dimensional array A is:

where each may be a scalar integer, an array of integers, or any other supported index. This includes () to select all indices within the entire dimension, ranges of the form or to select contiguous or strided subsections, and arrays of booleans to select elements at their indices.

If all the indices are scalars, then the result is a single element from the array . Otherwise, is an array with the same number of dimensions as the sum of the dimensionalities of all the indices.

If all indices are vectors, for example, then the shape of would be , with location of containing the value . If is changed to a two-dimensional matrix, then becomes an -dimensional array of shape . The matrix adds a dimension. The location contains the value at . All dimensions indexed with scalars are dropped. For example, the result of is an array with size . Its th element is populated by .

As a special part of this syntax, the keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the innermost array being indexed. Indexing syntax without the keyword is equivalent to a call to :

Example:

Empty ranges of the form are sometimes used to indicate the inter-index location between and . For example, the function uses this convention to indicate the insertion point of a value not found in a sorted array:

Assignment

The general syntax for assigning values in an n-dimensional array A is:

where each may be a scalar integer, an array of integers, or any other supported index. This includes () to select all indices within the entire dimension, ranges of the form or to select contiguous or strided subsections, and arrays of booleans to select elements at their indices.

If is an array, it must have the same number of elements as the product of the lengths of the indices: . The value in location of is overwritten with the value . If is not an array, its value is written to all referenced locations of .

Just as in Indexing, the keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the array being assigned into. Indexed assignment syntax without the keyword is equivalent to a call to :

Example:

Supported index types

In the expression , each may be a scalar index, an array of scalar indices, or an object that represents an array of scalar indices and can be converted to such by :

1. A scalar index. By default this includes:
   - Non-boolean integers
   - s, which behave like an -tuple of integers spanning multiple dimensions (see below for more details)
2. An array of scalar indices. This includes:
   - Vectors and multidimensional arrays of integers
   - Empty arrays like , which select no elements
   - s of the form or , which select contiguous or strided subsections from to (inclusive)
   - Any custom array of scalar indices that is a subtype of
   - Arrays of (see below for more details)
3. An object that represents an array of scalar indices and can be converted to such by . By default this includes:
   - (), which represents all indices within an entire dimension or across the entire array
   - Arrays of booleans, which select elements at their indices (see below for more details)

Cartesian indices

The special object represents a scalar index that behaves like an -tuple of integers spanning multiple dimensions. For example:

Considered alone, this may seem relatively trivial; simply gathers multiple integers together into one object that represents a single multidimensional index. When combined with other indexing forms and iterators that yield es, however, this can lead directly to very elegant and efficient code. See Iteration below, and for some more advanced examples, see this blog post on multidimensional algorithms and iteration.

Arrays of are also supported. They represent a collection of scalar indices that each span dimensions, enabling a form of indexing that is sometimes referred to as pointwise indexing. For example, it enables accessing the diagonal elements from the first "page" of from above:

This can be expressed much more simply with dot broadcasting and by combining it with a normal integer index (instead of extracting the first from as a separate step). It can even be combined with a to extract both diagonals from the two pages at the same time:

Warning

and arrays of are not compatible with the keyword to represent the last index of a dimension. Do not use in indexing expressions that may contain either or arrays thereof.

Logical indexing

Often referred to as logical indexing or indexing with a logical mask, indexing by a boolean array selects elements at the indices where its values are . Indexing by a boolean vector is effectively the same as indexing by the vector of integers that is returned by . Similarly, indexing by a -dimensional boolean array is effectively the same as indexing by the vector of s where its values are . A logical index must be a vector of the same length as the dimension it indexes into, or it must be the only index provided and match the size and dimensionality of the array it indexes into. It is generally more efficient to use boolean arrays as indices directly instead of first calling .

Iteration

The recommended ways to iterate over a whole array are

The first construct is used when you need the value, but not index, of each element. In the second construct, will be an if is an array type with fast linear indexing; otherwise, it will be a :

In contrast with , iterating with provides an efficient way to iterate over any array type.

Array traits

If you write a custom type, you can specify that it has fast linear indexing using

This setting will cause iteration over a to use integers. If you don't specify this trait, the default value is used.

Array and Vectorized Operators and Functions

The following operators are supported for arrays:

1. Unary arithmetic – ,
2. Binary arithmetic – , , , , ,
3. Comparison – , , (),

Most of the binary arithmetic operators listed above also operate elementwise when one argument is scalar: , , and when either argument is scalar, and and when the denominator is scalar. For example, and .

Additionally, to enable convenient vectorization of mathematical and other operations, Julia provides the dot syntax , e.g. or , for elementwise operations over arrays or mixtures of arrays and scalars (a Broadcasting operation); these have the additional advantage of "fusing" into a single loop when combined with other dot calls, e.g. .

Also, *every* binary operator supports a dot version that can be applied to arrays (and combinations of arrays and scalars) in such fused broadcasting operations, e.g. .

Note that comparisons such as operate on whole arrays, giving a single boolean answer. Use dot operators like for elementwise comparisons. (For comparison operations like , *only* the elementwise version is applicable to arrays.)

Also notice the difference between , which s elementwise over and , and , which finds the largest value within . The same relationship holds for and .

Broadcasting

It is sometimes useful to perform element-by-element binary operations on arrays of different sizes, such as adding a vector to each column of a matrix. An inefficient way to do this would be to replicate the vector to the size of the matrix:

This is wasteful when dimensions get large, so Julia offers , which expands singleton dimensions in array arguments to match the corresponding dimension in the other array without using extra memory, and applies the given function elementwise:

Dotted operators such as and are equivalent to calls (except that they fuse, as described below). There is also a function to specify an explicit destination (which can also be accessed in a fusing fashion by assignment), and functions and that broadcast the indices before indexing. Moreover, is equivalent to , providing a convenient syntax to broadcast any function (dot syntax). Nested ”dot calls” (including calls to etcetera) automatically fuse into a single call.

Additionally, is not limited to arrays (see the function documentation), it also handles tuples and treats any argument that is not an array, tuple or (except for) as a ”scalar”.

Implementation

The base array type in Julia is the abstract type . It is parametrized by the number of dimensions and the element type . and are aliases for the 1-d and 2-d cases. Operations on objects are defined using higher level operators and functions, in a way that is independent of the underlying storage. These operations generally work correctly as a fallback for any specific array implementation.

The type includes anything vaguely array-like, and implementations of it might be quite different from conventional arrays. For example, elements might be computed on request rather than stored. However, any concrete type should generally implement at least (returning an tuple), and ; mutable arrays should also implement . It is recommended that these operations have nearly constant time complexity, or technically Õ(1) complexity, as otherwise some array functions may be unexpectedly slow. Concrete types should also typically provide a method, which is used to allocate a similar array for and other out-of-place operations. No matter how an is represented internally, is the type of object returned by *integer* indexing (, when is not empty) and should be the length of the tuple returned by .

 is an abstract subtype of intended to include all arrays that are laid out at regular offsets in memory, and which can therefore be passed to external C and Fortran functions expecting this memory layout. Subtypes should provide a method that returns the ”stride” of dimension : increasing the index of dimension by should increase the index of by . If a pointer conversion method is provided, the memory layout should correspond in the same way to these strides.

The type is a specific instance of where elements are stored in column-major order (see additional notes in Performance Tips). and are aliases for the 1-d and 2-d cases. Specific operations such as scalar indexing, assignment, and a few other basic storage-specific operations are all that have to be implemented for , so that the rest of the array library can be implemented in a generic manner.

 is a specialization of that performs indexing by reference rather than by copying. A is created with the function, which is called the same way as (with an array and a series of index arguments). The result of looks the same as the result of , except the data is left in place. stores the input index vectors in a object, which can later be used to index the original array indirectly. By putting the macro in front of an expression or block of code, any slice in that expression will be converted to create a view instead.

 and are convenient aliases defined to make it possible for Julia to call a wider range of BLAS and LAPACK functions by passing them either or objects, and thus saving inefficiencies from memory allocation and copying.

The following example computes the QR decomposition of a small section of a larger array, without creating any temporaries, and by calling the appropriate LAPACK function with the right leading dimension size and stride parameters.

22.2 Sparse Vectors and Matrices

Julia has built-in support for sparse vectors and sparse matrices. Sparse arrays are arrays that contain enough zeros that storing them in a special data structure leads to savings in space and execution time, compared to dense arrays.

Compressed Sparse Column (CSC) Sparse Matrix Storage

In Julia, sparse matrices are stored in the Compressed Sparse Column (CSC) format. Julia sparse matrices have the type , where is the type of the stored values, and is the integer type for storing column pointers and row indices. The internal representation of is as follows:

The compressed sparse column storage makes it easy and quick to access the elements in the column of a sparse matrix, whereas accessing the sparse matrix by rows is considerably slower. Operations such as insertion of previously unstored entries one at a time in the CSC structure tend to be slow. This is because all elements of the sparse matrix that are beyond the point of insertion have to be moved one place over.

All operations on sparse matrices are carefully implemented to exploit the CSC data structure for performance, and to avoid expensive operations.

If you have data in CSC format from a different application or library, and wish to import it in Julia, make sure that you use 1-based indexing. The row indices in every column need to be sorted. If your object contains unsorted row indices, one quick way to sort them is by doing a double transpose.

In some applications, it is convenient to store explicit zero values in a . These *are* accepted by functions in (but there is no guarantee that they will be preserved in mutating operations). Such explicitly stored zeros are treated as structural nonzeros by many routines. The function returns the number of elements explicitly stored in the sparse data structure, including structural nonzeros. In order to count the exact number of numerical nonzeros, use , which inspects every stored element of a sparse matrix. , and the in-place , can be used to remove stored zeros from the sparse matrix.

Sparse Vector Storage

Sparse vectors are stored in a close analog to compressed sparse column format for sparse matrices. In Julia, sparse vectors have the type where is the type of the stored values and the integer type for the indices. The internal representation is as follows:

As for , the type can also contain explicitly stored zeros. (See Sparse Matrix Storage.).

Sparse Vector and Matrix Constructors

The simplest way to create sparse arrays is to use functions equivalent to the and functions that Julia provides for working with dense arrays. To produce sparse arrays instead, you can use the same names with an prefix:

The function is often a handy way to construct sparse arrays. For example, to construct a sparse matrix we can input a vector of row indices, a vector of column indices, and a vector of stored values (this is also known as the COO (coordinate) format). then constructs a sparse matrix such that . The equivalent sparse vector constructor is , which takes the (row) index vector and the vector with the stored values and constructs a sparse vector such that .

The inverse of the and functions is , which retrieves the inputs used to create the sparse array. There is also a function which only returns the index vectors.

Another way to create a sparse array is to convert a dense array into a sparse array using the function:

You can go in the other direction using the constructor. The function can be used to query if a matrix is sparse.

Sparse matrix operations

Arithmetic operations on sparse matrices also work as they do on dense matrices. Indexing of, assignment into, and concatenation of sparse matrices work in the same way as dense matrices. Indexing operations, especially assignment, are expensive, when carried out one element at a time. In many cases it may be better to convert the sparse matrix into format using , manipulate the values or the structure in the dense vectors , and then reconstruct the sparse matrix.

Correspondence of dense and sparse methods

The following table gives a correspondence between built-in methods on sparse matrices and their corresponding methods on dense matrix types. In general, methods that generate sparse matrices differ from their dense counterparts in that the resulting matrix follows the same sparsity pattern as a given sparse matrix , or that the resulting sparse matrix has density , i.e. each matrix element has a probability of being non-zero.

Details can be found in the Sparse Vectors and Matrices section of the standard library reference.

Sparse	Dense	Description
		Creates a *m*-by-*n* matrix of zeros. (is empty.)
		Creates a matrix filled with ones. Unlike the dense version, has the same sparsity pattern as *S*.
		Creates a *n*-by-*n* identity matrix.
		Interconverts between dense and sparse formats.
		Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed uniformly on the half-open interval $[0, 1)$.
		Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed according to the standard normal (Gaussian) distribution.
		Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed according to the *X* distribution. (Requires the package.)

Chapter 23

Linear algebra

In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations. Basic operations, such as , , and are all supported:

As well as other useful operations, such as finding eigenvalues or eigenvectors:

In addition, Julia provides many factorizations which can be used to speed up problems such as linear solve or matrix exponentiation by pre-factorizing a matrix into a form more amenable (for performance or memory reasons) to the problem. See the documentation on for more information. As an example:

Since is not Hermitian, symmetric, triangular, tridiagonal, or bidiagonal, an LU factorization may be the best we can do. Compare with:

Here, Julia was able to detect that is in fact symmetric, and used a more appropriate factorization. Often it's possible to write more efficient code for a matrix that is known to have certain properties e.g. it is symmetric, or tridiagonal. Julia provides some special types so that you can "tag" matrices as having these properties. For instance:

has been tagged as a matrix that's (real) symmetric, so for later operations we might perform on it, such as eigenfactorization or computing matrix-vector products, efficiencies can be found by only referencing half of it. For example:

The operation here performs the linear solution. Julia's parser provides convenient dispatch to specialized methods for the *transpose* of a matrix left-divided by a vector, or for the various combinations of transpose operations in matrix-matrix solutions. Many of these are further specialized for certain special matrix types. For example, will end up calling while will end up calling , even though we used the same left-division operator. This works for matrices too: would call . The left-division operator is pretty powerful and it's easy to write compact, readable code that is flexible enough to solve all sorts of systems of linear equations.

23.1 Special matrices

Matrices with special symmetries and structures arise often in linear algebra and are frequently associated with various matrix factorizations. Julia features a rich collection of special matrix types, which allow for fast computation with specialized routines that are specially developed for particular matrix types.

The following tables summarize the types of special matrices that have been implemented in Julia, as well as whether hooks to various optimized methods for them in LAPACK are available.

Elementary operations

Legend:

Type	Description
	Symmetric matrix
	Hermitian matrix
	Upper triangular matrix
	Lower triangular matrix
	Tridiagonal matrix
	Symmetric tridiagonal matrix
	Upper/lower bidiagonal matrix
	Diagonal matrix
	Uniform scaling operator

Matrix type					Other functions with optimized methods
				MV	, ,
				MV	, ,
			MV	MV	,
			MV	MV	,
	M	M	MS	MV	,
	M	M	MS	MV	
	M	M	MS	MV	
	M	M	MV	MV	, , ,
	M	M	MVS	MVS	

Key	Description
M (matrix)	An optimized method for matrix-matrix operations is available
V (vector)	An optimized method for matrix-vector operations is available
S (scalar)	An optimized method for matrix-scalar operations is available

Matrix factorizations

Legend:

The uniform scaling operator

A operator represents a scalar times the identity operator, . The identity operator is defined as a constant and is an instance of . The size of these operators are generic and match the other matrix in the binary operations , , and . For and this means that must be square. Multiplication with the identity operator is a noop (except for checking that the scaling factor is one) and therefore almost without overhead.

23.2 Matrix factorizations

Matrix factorizations (a.k.a. matrix decompositions) compute the factorization of a matrix into a product of matrices, and are one of the central concepts in linear algebra.

The following table summarizes the types of matrix factorizations that have been implemented in Julia. Details of their associated methods can be found in the Linear Algebra section of the standard library documentation.

Matrix type	LAPACK					
	SY		ARI			
	HE		ARI			
	TR	A	A	A		
	TR	A	A	A		
	ST	A	ARI	AV		
	GT					
	BD				A	A
	DI		A			

Key	Description	Example
A (all)	An optimized method to find all the characteristic values and/or vectors is available	e.g.
R (range)	An optimized method to find the th through the th characteristic values are available	
I (interval)	An optimized method to find the characteristic values in the interval [,] is available	
V (vectors)	An optimized method to find the characteristic vectors corresponding to the characteristic values is available	

Type	Description
	Cholesky factorization
	Pivoted Cholesky factorization
	LU factorization
	LU factorization for matrices
	LU factorization for sparse matrices (computed by UMFPack)
	QR factorization
	Compact WY form of the QR factorization
	Pivoted QR factorization
	Hessenberg decomposition
	Spectral decomposition
	Singular value decomposition
	Generalized SVD

Chapter 24

Networking and Streams

Julia provides a rich interface to deal with streaming I/O objects such as terminals, pipes and TCP sockets. This interface, though asynchronous at the system level, is presented in a synchronous manner to the programmer and it is usually unnecessary to think about the underlying asynchronous operation. This is achieved by making heavy use of Julia cooperative threading (coroutine) functionality.

24.1 Basic Stream I/O

All Julia streams expose at least a and a method, taking the stream as their first argument, e.g.:

Note that returns 11, the number of bytes (in) written to , but this return value is suppressed with the .

Here Enter was pressed again so that Julia would read the newline. Now, as you can see from this example, takes the data to write as its second argument, while takes the type of the data to be read as the second argument.

For example, to read a simple byte array, we could do:

However, since this is slightly cumbersome, there are several convenience methods provided. For example, we could have written the above as:

or if we had wanted to read the entire line instead:

Note that depending on your terminal settings, your TTY may be line buffered and might thus require an additional enter before the data is sent to Julia.

To read every line from you can use :

or if you wanted to read by character instead:

24.2 Text I/O

Note that the method mentioned above operates on binary streams. In particular, values do not get converted to any canonical text representation but are written out as is:

Note that is written to by the function and that the returned value is (since is one byte).

For text I/O, use the or methods, depending on your needs (see the standard library reference for a detailed discussion of the difference between the two):

24.3 IO Output Contextual Properties

Sometimes IO output can benefit from the ability to pass contextual information into show methods. The object provides this framework for associating arbitrary metadata with an IO object. For example, adds a hinting parameter to the IO object that the invoked show method should print a shorter output (if applicable).

24.4 Working with Files

Like many other environments, Julia has an function, which takes a filename and returns an object that you can use to read and write things from the file. For example if we have a file, , whose contents are :

If you want to write to a file, you can open it with the write () flag:

If you examine the contents of at this point, you will notice that it is empty; nothing has actually been written to disk yet. This is because the must be closed before the write is actually flushed to disk:

Examining again will show its contents have been changed.

Opening a file, doing something to its contents, and closing it again is a very common pattern. To make this easier, there exists another invocation of which takes a function as its first argument and filename as its second, opens the file, calls the function with the file as an argument, and then closes it again. For example, given a function:

You can call:

to open , call , close and return the capitalized contents.

To avoid even having to define a named function, you can use the syntax, which creates an anonymous function on the fly:

24.5 A simple TCP example

Let's jump right in with a simple example involving TCP sockets. Let's first create a simple server:

To those familiar with the Unix socket API, the method names will feel familiar, though their usage is somewhat simpler than the raw Unix socket API. The first call to will create a server waiting for incoming connections on the specified port (2000) in this case. The same function may also be used to create various other kinds of servers:

Note that the return type of the last invocation is different. This is because this server does not listen on TCP, but rather on a named pipe (Windows) or UNIX domain socket. Also note that Windows named pipe format has to be a specific pattern such that the name prefix () uniquely identifies the file type. The difference between TCP and named pipes or UNIX domain sockets is subtle and has to do with the and methods. The method retrieves a connection to the client that is connecting on the server we just created, while the function connects to a server using the specified method. The function takes the same arguments as , so, assuming the environment (i.e. host, cwd, etc.) is the same you should be able to pass the same arguments to as you did to listen to establish the connection. So let's try that out (after having created the server above):

As expected we saw "Hello World" printed. So, let's actually analyze what happened behind the scenes. When we called , we connect to the server we had just created. Meanwhile, the accept function returns a server-side connection to the newly created socket and prints "Hello World" to indicate that the connection was successful.

A great strength of Julia is that since the API is exposed synchronously even though the I/O is actually happening asynchronously, we didn't have to worry callbacks or even making sure that the server gets to run. When we called the current task waited for the connection to be established and only continued executing after that was done. In this pause, the server task resumed execution (because a connection request was now available), accepted the connection, printed the message and waited for the next client. Reading and writing works in the same way. To see this, consider the following simple echo server:

As with other streams, use to disconnect the socket:

24.6 Resolving IP Addresses

One of the methods that does not follow the methods is , which will attempt to connect to the host given by the parameter on the port given by the port parameter. It allows you to do things like:

At the base of this functionality is , which will do the appropriate address resolution:

Chapter 25

Parallel Computing

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly. There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node). Perhaps more surprisingly, similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache. Consequently, a good multiprocessing environment should allow control over the "ownership" of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

Julia's implementation of message passing is different from other environments such as MPI [1]. Communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one process in a two-process operation. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions.

Parallel programming in Julia is built on two primitives: *remote references* and *remote calls.* A remote reference is an object that can be used from any process to refer to an object stored on a particular process. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

Remote references come in two flavors: and .

A remote call returns a to its result. Remote calls return immediately; the process that made the call proceeds to its next operation while the remote call happens somewhere else. You can wait for a remote call to finish by calling on the returned , and you can obtain the full value of the result using .

On the other hand, s are rewritable. For example, multiple processes can co-ordinate their processing by referencing the same remote .

Each process has an associated identifier. The process providing the interactive Julia prompt always has an equal to 1. The processes used by default for parallel operations are referred to as "workers". When there is only one process, process 1 is considered a worker. Otherwise, workers are considered to be all processes other than process 1.

Let's try this out. Starting with provides worker processes on the local machine. Generally it makes sense for to equal the number of CPU cores on the machine.

The first argument to is the function to call. Most parallel programming in Julia does not reference specific processes or the number of processes available, but is considered a low-level interface providing finer control. The second argument to is the of the process that will do the work, and the remaining arguments will be passed to the function being called.

As you can see, in the first line we asked process 2 to construct a 2-by-2 random matrix, and in the second line we asked it to add 1 to it. The result of both calculations is available in the two futures, and . The macro evaluates the expression in the second argument on the process specified by the first argument.

Occasionally you might want a remotely-computed value immediately. This typically happens when you read from a remote object to obtain data needed by the next local operation. The function exists for this purpose. It is equivalent to but is more efficient.

Remember that is equivalent to , so this call fetches the first element of the future .

The syntax of is not especially convenient. The macro makes things easier. It operates on an expression rather than a function, and picks where to do the operation for you:

Note that we used instead of . This is because we do not know where the code will run, so in general a might be required to move to the process doing the addition. In this case, is smart enough to perform the computation on the process that owns , so the will be a no-op (no work is done).

(It is worth noting that is not built-in but defined in Julia as a macro. It is possible to define your own such constructs.)

An important thing to remember is that, once fetched, a will cache its value locally. Further calls do not entail a network hop. Once all referencing s have fetched, the remote stored value is deleted.

25.1 Code Availability and Loading Packages

Your code must be available on any process that runs it. For example, type the following into the Julia prompt:

Process 1 knew about the function , but process 2 did not.

Most commonly you'll be loading code from files or packages, and you have a considerable amount of flexibility in controlling which processes load code. Consider a file, , containing the following code:

Starting Julia with , you can use this to verify the following:

- loads the file on just a single process (whichever one executes the statement).
- causes the module to be loaded on all processes; however, the module is brought into scope only on the one executing the statement.
- As long as is loaded on process 2, commands like

  allow you to store an object of type on process 2 even if is not in scope on process 2.

You can force a command to run on all processes using the macro. For example, can also be used to directly define a function on all processes:

A file can also be preloaded on multiple processes at startup, and a driver script can be used to drive the computation:

The Julia process running the driver script in the example above has an equal to 1, just like a process providing an interactive prompt.

The base Julia installation has in-built support for two types of clusters:

- A local cluster specified with the option as shown above.
- A cluster spanning machines using the option. This uses a passwordless login to start Julia worker processes (from the same path as the current host) on the specified machines.

Functions , , , and others are available as a programmatic means of adding, removing and querying the processes in a cluster.

Note that workers do not run a startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

Other types of clusters can be supported by writing your own custom , as described below in the ClusterManagers section.

25.2 Data Movement

Sending messages and moving data constitute most of the overhead in a parallel program. Reducing the number of messages and the amount of data sent is critical to achieving performance and scalability. To this end, it is important to understand the data movement performed by Julia's various parallel programming constructs.

 can be considered an explicit data movement operation, since it directly asks that an object be moved to the local machine. (and a few related constructs) also moves data, but this is not as obvious, hence it can be called an implicit data movement operation. Consider these two approaches to constructing and squaring a random matrix:

Method 1:

```

```

Method 2:

```

```

The difference seems trivial, but in fact is quite significant due to the behavior of . In the first method, a random matrix is constructed locally, then sent to another process where it is squared. In the second method, a random matrix is both constructed and squared on another process. Therefore the second method sends much less data than the first.

In this toy example, the two methods are easy to distinguish and choose from. However, in a real program designing data movement might require more thought and likely some measurement. For example, if the first process needs matrix then the first method might be better. Or, if computing is expensive and only the current process has it, then moving it

to another process might be unavoidable. Or, if the current process has very little to do between the and , it might be better to eliminate the parallelism altogether. Or imagine is replaced with a more expensive operation. Then it might make sense to add another statement just for this step.

Chapter 26

Global variables

Expressions executed remotely via , or closures specified for remote execution using may refer to global variables. Global bindings under module are treated a little differently compared to global bindings in other modules. Consider the following code snippet:

In this case is a function that takes 2D array as a parameter, and MUST be defined in the remote process. You could use any function other than as long as it is defined in the remote process and accepts the appropriate parameter.

Note that is a global variable defined in the local workspace. Worker 2 does not have a variable called under . The act of shipping the closure to worker 2 results in being defined on 2. continues to exist on worker 2 even after the call returns. Remote calls with embedded global references (under module only) manage globals as follows:

- New global bindings are created on destination workers if they are referenced as part of a remote call.
- Global constants are declared as constants on remote nodes too.
- Globals are re-sent to a destination worker only in the context of a remote call, and then only if its value has changed. Also, the cluster does not synchronize global bindings across nodes. For example:

  Executing the above snippet results in on worker 2 having a different value from on worker 3, while the value of on node 1 is set to .

As you may have realized, while memory associated with globals may be collected when they are reassigned on the master, no such action is taken on the workers as the bindings continue to be valid. can be used to manually reassign specific globals on remote nodes to once they are no longer required. This will release any memory associated with them as part of a regular garbage collection cycle.

Thus programs should be careful referencing globals in remote calls. In fact, it is preferable to avoid them altogether if possible. If you must reference globals, consider using blocks to localize global variables.

For example:

As can be seen, global variable is defined on worker 2, but is captured as a local variable and hence a binding for does not exist on worker 2.

26.1 Parallel Map and Loops

Fortunately, many useful parallel computations do not require data movement. A common example is a Monte Carlo simulation, where multiple processes can handle independent simulation trials simultaneously. We can use to flip coins on two processes. First, write the following function in :

The function simply adds together random bits. Here is how we can perform some trials on two machines, and add together the results:

This example demonstrates a powerful and often-used parallel programming pattern. Many iterations run independently over several processes, and then their results are combined using some function. The combination process is called a *reduction*, since it is generally tensor-rank-reducing: a vector of numbers is reduced to a single number, or a matrix is reduced to a single row or column, etc. In code, this typically looks like the pattern , where is the accumulator,

is the reduction function, and the are the elements being reduced. It is desirable for to be associative, so that it does not matter what order the operations are performed in.

Notice that our use of this pattern with can be generalized. We used two explicit statements, which limits the parallelism to two processes. To run on any number of processes, we can use a *parallel for loop*, which can be written in Julia using like this:

This construct implements the pattern of assigning iterations to multiple processes, and combining them with a specified reduction (in this case). The result of each iteration is taken as the value of the last expression inside the loop. The whole parallel loop expression itself evaluates to the final answer.

Note that although parallel for loops look like serial for loops, their behavior is dramatically different. In particular, the iterations do not happen in a specified order, and writes to variables or arrays will not be globally visible since iterations run on different processes. Any variables used inside the parallel loop will be copied and broadcast to each process.

For example, the following code will not work as intended:

This code will not initialize all of , since each process will have a separate copy of it. Parallel for loops like these must be avoided. Fortunately, Shared Arrays can be used to get around this limitation:

Using ”outside” variables in parallel loops is perfectly reasonable if the variables are read-only:

Here each iteration applies to a randomly-chosen sample from a vector shared by all processes.

As you could see, the reduction operator can be omitted if it is not needed. In that case, the loop executes asynchronously, i.e. it spawns independent tasks on all available workers and returns an array of immediately without waiting for completion. The caller can wait for the completions at a later point by calling on them, or wait for completion at the end of the loop by prefixing it with , like .

In some cases no reduction operator is needed, and we merely wish to apply a function to all integers in some range (or, more generally, to all elements in some collection). This is another useful operation called *parallel map*, implemented in Julia as the function. For example, we could compute the singular values of several large random matrices in parallel as follows:

Julia's is designed for the case where each function call does a large amount of work. In contrast, can handle situations where each iteration is tiny, perhaps merely summing two numbers. Only worker processes are used by both and for the parallel computation. In case of , the final reduction is done on the calling process.

26.2 Synchronization With Remote References

26.3 Scheduling

Julia's parallel programming platform uses Tasks (aka Coroutines) to switch among multiple computations. Whenever code performs a communication operation like or , the current task is suspended and a scheduler picks another task to run. A task is restarted when the event it is waiting for completes.

For many problems, it is not necessary to think about tasks directly. However, they can be used to wait for multiple events at the same time, which provides for *dynamic scheduling*. In dynamic scheduling, a program decides what to compute or where to compute it based on when other jobs finish. This is needed for unpredictable or unbalanced workloads, where we want to assign more work to processes only when they finish their current tasks.

As an example, consider computing the singular values of matrices of different sizes:

If one process handles both 800×800 matrices and another handles both 600×600 matrices, we will not get as much scalability as we could. The solution is to make a local task to "feed" work to each process when it completes its current task. For example, consider a simple implementation:

```

```

is similar to , but only runs tasks on the local process. We use it to create a ”feeder” task for each process. Each task picks the next index that needs to be computed, then waits for its process to finish, then repeats until we run out of indexes. Note that the feeder tasks do not begin to execute until the main task reaches the end of the block, at which point it surrenders control and waits for all the local tasks to complete before returning from the function. The feeder tasks are able to share state via because they all run on the same process. No locking is required, since the threads are scheduled cooperatively and not preemptively. This means context switches only occur at well-defined points: in this case, when is called.

26.4 Channels

The section on s in Control Flow discussed the execution of multiple functions in a co-operative manner. s can be quite useful to pass data between running tasks, particularly those involving I/O operations.

Examples of operations involving I/O include reading/writing to files, accessing web services, executing external programs, etc. In all these cases, overall execution time can be improved if other tasks can be run while a file is being read, or while waiting for an external service/program to complete.

A channel can be visualized as a pipe, i.e., it has a write end and read end.

- Multiple writers in different tasks can write to the same channel concurrently via calls.
- Multiple readers in different tasks can read data concurrently via calls.
- As an example:

```

```

- Channels are created via the constructor. The channel will only hold objects of type . If the type is not specified, the channel can hold objects of any type. refers to the maximum number of elements that can be held in the channel at any time. For example, creates a channel that can hold a maximum of 32 objects of any type. A can hold up to 64 objects of at any time.
- If a is empty, readers (on a call) will block until data is available.

- If a is full, writers (on a call) will block until space becomes available.
- tests for the presence of any object in the channel, while waits for an object to become available.
- A is in an open state initially. This means that it can be read from and written to freely via and calls. closes a . On a closed , will fail. For example:

- and (which retrieves but does not remove the value) on a closed channel successfully return any existing values until it is emptied. Continuing the above example:

A can be used as an iterable object in a loop, in which case the loop runs as long as the has data or is open. The loop variable takes on all values added to the . The loop is terminated once the is closed and emptied.

For example, the following would cause the loop to wait for more data:

while this will return after reading all data:

Consider a simple example using channels for inter-task communication. We start 4 tasks to process data from a single channel. Jobs, identified by an id (), are written to the channel. Each task in this simulation reads a , waits for a random amout of time and writes back a tuple of and the simulated time to the results channel. Finally all the are printed out.

The current version of Julia multiplexes all tasks onto a single OS thread. Thus, while tasks involving I/O operations benefit from parallel execution, compute bound tasks are effectively executed sequentially on a single OS thread. Future versions of Julia may support scheduling of tasks on multiple threads, in which case compute bound tasks will see benefits of parallel execution too.

26.5 Remote References and AbstractChannels

Remote references always refer to an implementation of an .

A concrete implementation of an (like), is required to implement , , , and . The remote object referred to by a is stored in a , i.e., a of size 1 capable of holding objects of type.

, which is rewritable, can point to any type and size of channels, or any other implementation of an .

The constructor allows us to construct references to channels holding more than one value of a specific type. is a function executed on and it must return an .

For example, , will return a reference to a channel of type and size 10. The channel exists on worker .

Methods , , , and on a are proxied onto the backing store on the remote process.

can thus be used to refer to user implemented objects. A simple example of this is provided in which uses a dictionary as its remote store.

26.6 Channels and RemoteChannels

- A is local to a process. Worker 2 cannot directly refer to a on worker 3 and vice-versa. A , however, can put and take values across workers.
- A can be thought of as a *handle* to a .
- The process id, , associated with a identifies the process where the backing store, i.e., the backing exists.
- Any process with a reference to a can put and take items from the channel. Data is automatically sent to (or retrieved from) the process a is associated with.
- Serializing a also serializes any data present in the channel. Deserializing it therefore effectively makes a copy of the original object.
- On the other hand, serializing a only involves the serialization of an identifier that identifies the location and instance of referred to by the handle. A deserialized object (on any worker), therefore also points to the same backing store as the original.

The channels example from above can be modified for interprocess communication, as shown below.

We start 4 workers to process a single remote channel. Jobs, identified by an id (), are written to the channel. Each remotely executing task in this simulation reads a , waits for a random amount of time and writes back a tuple of , time taken and its own to the results channel. Finally all the are printed out on the master process.

26.7 Remote References and Distributed Garbage Collection

Objects referred to by remote references can be freed only when *all* held references in the cluster are deleted.

The node where the value is stored keeps track of which of the workers have a reference to it. Every time a or a (unfetched) is serialized to a worker, the node pointed to by the reference is notified. And every time a or a (unfetched) is garbage collected locally, the node owning the value is again notified. This is implemented in an internal cluster aware serializer. Remote references are only valid in the context of a running cluster. Serializing and deserializing references to and from regular objects is not supported.

The notifications are done via sending of ”tracking” messages–an ”add reference” message when a reference is serialized to a different process and a ”delete reference” message when a reference is locally garbage collected.

Since s are write-once and cached locally, the act of ing a also updates reference tracking information on the node owning the value.

The node which owns the value frees it once all references to it are cleared.

With s, serializing an already fetched to a different node also sends the value since the original remote store may have collected the value by this time.

It is important to note that *when* an object is locally garbage collected depends on the size of the object and the current memory pressure in the system.

In case of remote references, the size of the local reference object is quite small, while the value stored on the remote node may be quite large. Since the local object may not be collected immediately, it is a good practice to explicitly call on local instances of a , or on unfetched s. Since calling on a also removes its reference from the remote store, this is not required on fetched s. Explicitly calling results in an immediate message sent to the remote node to go ahead and remove its reference to the value.

Once finalized, a reference becomes invalid and cannot be used in any further calls.

26.8 Shared Arrays

Shared Arrays use system shared memory to map the same array across many processes. While there are some similarities to a , the behavior of a is quite different. In a , each process has local access to just a chunk of the data, and no two processes share the same chunk; in contrast, in a each ”participating” process has access to the entire array. A is a good choice when you want to have a large amount of data jointly accessible to two or more processes on the same machine.

indexing (assignment and accessing values) works just as with regular arrays, and is efficient because the underlying memory is available to the local process. Therefore, most algorithms work naturally on s, albeit in single-process mode. In cases where an algorithm insists on an input, the underlying array can be retrieved from a by calling . For other types, just returns the object itself, so it’s safe to use on any -type object.

The constructor for a shared array is of the form:

which creates an -dimensional shared array of a bits type and size across the processes specified by . Unlike distributed arrays, a shared array is accessible only from those participating workers specified by the named argument (and the creating process too, if it is on the same host).

If an function, of signature , is specified, it is called on all the participating workers. You can specify that each worker runs the function on a distinct portion of the array, thereby parallelizing initialization.

Here’s a brief example:

provides disjoint one-dimensional ranges of indexes, and is sometimes convenient for splitting up tasks among processes. You can, of course, divide the work any way you wish:

Since all processes have access to the underlying data, you do have to be careful not to set up conflicts. For example:

would result in undefined behavior. Because each process fills the *entire* array with its own , whichever process is the last to execute (for any particular element of) will have its retained.

As a more extended and complex example, consider running the following "kernel" in parallel:

In this case, if we try to split up the work using a one-dimensional index, we are likely to run into trouble: if is near the end of the block assigned to one worker and is near the beginning of the block assigned to another, it's very likely that will not be ready at the time it's needed for computing . In such cases, one is better off chunking the array manually. Let's split along the second dimension. Define a function that returns the indexes assigned to this worker:

Next, define the kernel:

We also define a convenience wrapper for a implementation

Now let's compare three different versions, one that runs in a single process:

one that uses :

and one that delegates in chunks:

If we create s and time these functions, we get the following results (with):

Run the functions once to JIT-compile and them on the second run:

The biggest advantage of is that it minimizes traffic among the workers, allowing each to compute for an extended time on the assigned piece.

26.9 Shared Arrays and Distributed Garbage Collection

Like remote references, shared arrays are also dependent on garbage collection on the creating node to release references from all participating workers. Code which creates many short lived shared array objects would benefit from explicitly finalizing these objects as soon as possible. This results in both memory and file handles mapping the shared segment being released sooner.

26.10 ClusterManagers

The launching, management and networking of Julia processes into a logical cluster is done via cluster managers. A is responsible for

- launching worker processes in a cluster environment
- managing events during the lifetime of each worker
- optionally, providing data transport

A Julia cluster has the following characteristics:

- The initial Julia process, also called the , is special and has an of 1.
- Only the process can add or remove worker processes.
- All processes can directly communicate with each other.

Connections between workers (using the in-built TCP/IP transport) is established in the following manner:

- is called on the master process with a object.
- calls the appropriate method which spawns required number of worker processes on appropriate machines.
- Each worker starts listening on a free port and writes out its host and port information to .
- The cluster manager captures the of each worker and makes it available to the master process.

- The master process parses this information and sets up TCP/IP connections to each worker.
- Every worker is also notified of other workers in the cluster.
- Each worker connects to all workers whose is less than the worker's own .
- In this way a mesh network is established, wherein every worker is directly connected with every other worker.

While the default transport layer uses plain , it is possible for a Julia cluster to provide its own transport.

Julia provides two in-built cluster managers:

- , used when or are called
- , used when is called with a list of hostnames

 is used to launch additional workers on the same host, thereby leveraging multi-core and multi-processor hardware.

Thus, a minimal cluster manager would need to:

- be a subtype of the abstract
- implement , a method responsible for launching new workers
- implement , which is called at various events during a worker's lifetime (for example, sending an interrupt signal)

 requires to implement:

As an example let us see how the , the manager responsible for starting workers on the same host, is implemented:

The method takes the following arguments:

- : the cluster manager that is called with

- : all the keyword arguments passed to
- : the array to append one or more objects to
- : the condition variable to be notified as and when workers are launched

The method is called asynchronously in a separate task. The termination of this task signals that all requested workers have been launched. Hence the function MUST exit as soon as all the requested workers have been launched.

Newly launched workers are connected to each other and the master process in an all-to-all manner. Specifying the command line argument results in the launched processes initializing themselves as workers and connections being set up via TCP/IP sockets.

All workers in a cluster share the same cookie as the master. When the cookie is unspecified, i.e, with the option, the worker tries to read it from its standard input. and both pass the cookie to newly launched workers via their standard inputs.

By default a worker will listen on a free port at the address returned by a call to . A specific address to listen on may be specified by optional argument . This is useful for multi-homed hosts.

As an example of a non-TCP/IP transport, an implementation may choose to use MPI, in which case must NOT be specified. Instead, newly launched workers should call before using any of the parallel constructs.

For every worker launched, the method must add a object (with appropriate fields initialized) to

Most of the fields in are used by the inbuilt managers. Custom cluster managers would typically specify only or / :

- If is specified, it is used to read host/port information. A Julia worker prints out its bind address and port at startup. This allows Julia workers to listen on any free port available instead of requiring worker ports to be configured manually.

- If is not specified, and are used to connect.
- , and are relevant for launching additional workers from a worker. For example, a cluster manager may launch a single worker per node, and use that to launch additional workers.
  - with an integer value will launch a total of workers.
  - with a value of will launch as many workers as the number of cores on that machine.
  - is the name of the executable including the full path.
  - should be set to the required command line arguments for new workers.
- , , and are used when a ssh tunnel is required to connect to the workers from the master process.
- is provided for custom cluster managers to store their own worker-specific information.

is called at different times during the worker's lifetime with appropriate values:

- with / when a worker is added / removed from the Julia worker pool.
- with when is called. The should signal the appropriate worker with an interrupt signal.
- with for cleanup purposes.

26.11 Cluster Managers with Custom Transports

Replacing the default TCP/IP all-to-all socket connections with a custom transport layer is a little more involved. Each Julia process has as many communication tasks as the workers it is connected to. For example, consider a Julia cluster of 32 processes in an all-to-all mesh network:

- Each Julia process thus has 31 communication tasks.
- Each task handles all incoming messages from a single remote worker in a message-processing loop.
- The message-processing loop waits on an object (for example, a in the default implementation), reads an entire message, processes it and waits for the next one.
- Sending messages to a process is done directly from any Julia task–not just communication tasks–again, via the appropriate object.

Replacing the default transport requires the new implementation to set up connections to remote workers and to provide appropriate objects that the message-processing loops can wait on. The manager-specific callbacks to be implemented are:

The default implementation (which uses TCP/IP sockets) is implemented as .

should return a pair of objects, one for reading data sent from worker , and the other to write data that needs to be sent to worker . Custom cluster managers can use an in-memory as the plumbing to proxy data between the custom, possibly non- transport and Julia's in-built parallel infrastructure.

A is an in-memory which behaves like an –it is a stream which can be handled asynchronously.

Folder contains an example of using ZeroMQ to connect Julia workers in a star topology with a 0MQ broker in the middle. Note: The Julia processes are still all *logically* connected to each other–any worker can message any other worker directly without any awareness of 0MQ being used as the transport layer.

When using custom transports:

- Julia workers must NOT be started with . Starting with will result in the newly launched workers defaulting to the TCP/IP socket transport implementation.
- For every incoming logical connection with a worker, must be called. This launches a new task that handles reading and writing of messages from/to the worker represented by the objects.
- MUST be called as part of worker process initialization.
- Field in can be set by the cluster manager when is called. The value of this field is passed in in all callbacks. Typically, it carries information on *how to connect* to a worker. For example, the TCP/IP socket transport uses this field to specify the tuple at which to connect to a worker.

is called to remove a worker from the cluster. On the master process, the corresponding objects must be closed by the implementation to ensure proper cleanup. The default implementation simply executes an call on the specified remote worker.

is an example that shows a simple implementation using UNIX domain sockets for cluster setup.

26.12 Network Requirements for LocalManager and SSHManager

Julia clusters are designed to be executed on already secured environments on infrastructure such as local laptops, departmental clusters, or even the cloud. This section covers network security requirements for the inbuilt and :

- The master process does not listen on any port. It only connects out to the workers.
- Each worker binds to only one of the local interfaces and listens on an ephemeral port number assigned by the OS.
- , used by , by default binds only to the loopback interface. This means that workers started later on remote hosts (or by anyone with malicious intentions) are unable to connect to the cluster. An followed by an will fail. Some users may need to create a cluster comprising their local system and a few remote systems. This can be done by explicitly requesting to bind to an external network interface via the keyword argument: .
- , used by , launches workers on remote hosts via SSH. By default SSH is only used to launch Julia workers. Subsequent master-worker and worker-worker connections use plain, unencrypted TCP/IP sockets. The remote hosts must have passwordless login enabled. Additional SSH flags or credentials may be specified via keyword argument .
- is useful when we wish to use SSH connections for master-worker too. A typical scenario for this is a local laptop running the Julia REPL (i.e., the master) with the rest of the cluster on the cloud, say on Amazon EC2. In this case only port 22 needs to be opened at the remote cluster coupled with SSH client authenticated via public key infrastructure (PKI). Authentication credentials can be supplied via , for example .

  In an all-to-all topology (the default), all workers connect to each other via plain TCP sockets. The security policy on the cluster nodes must thus ensure free connectivity between workers for the ephemeral port range (varies by OS).

  Securing and encrypting all worker-worker traffic (via SSH) or encrypting individual messages can be done via a custom ClusterManager.

26.13 Cluster Cookie

All processes in a cluster share the same cookie which, by default, is a randomly generated string on the master process:

- returns the cookie, while sets it and returns the new cookie.
- All connections are authenticated on both sides to ensure that only workers started by the master are allowed to connect to each other.
- The cookie may be passed to the workers at startup via argument . If argument is specified without the cookie, the worker tries to read the cookie from its standard input (STDIN). The STDIN is closed immediately after the cookie is retrieved.
- ClusterManagers can retrieve the cookie on the master by calling . Cluster managers not using the default TCP/IP transport (and hence not specifying) must call with the same cookie as on the master.

Note that environments requiring higher levels of security can implement this via a custom . For example, cookies can be pre-shared and hence not specified as a startup argument.

26.14 Specifying Network Topology (Experimental)

The keyword argument passed to is used to specify how the workers must be connected to each other:

- , the default: all workers are connected to each other.
- : only the driver process, i.e. 1, has connections to the workers.
- : the method of the cluster manager specifies the connection topology via the fields and in . A worker with a cluster-manager-provided identity will connect to all workers specified in .

Keyword argument only affects option . If , the cluster starts off with the master connected to all workers. Specific worker-worker connections are established at the first remote invocation between two workers. This helps in reducing initial resources allocated for intra-cluster communication. Connections are setup depending on the runtime requirements of a parallel program. Default value for is .

Currently, sending a message between unconnected workers results in an error. This behaviour, as with the functionality and interface, should be considered experimental in nature and may change in future releases.

26.15 Multi-Threading (Experimental)

In addition to tasks, remote calls, and remote references, Julia from forwards will natively support multi-threading. Note that this section is experimental and the interfaces may change in the future.

Setup

By default, Julia starts up with a single thread of execution. This can be verified by using the command :

The number of threads Julia starts up with is controlled by an environment variable called . Now, let's start up Julia with 4 threads:

(The above command works on bourne shells on Linux and OSX. Note that if you're using a C shell on these platforms, you should use the keyword instead of . If you're on Windows, start up the command line in the location of and use instead of .)

Let's verify there are 4 threads at our disposal.

But we are currently on the master thread. To check, we use the command

The Macro

Let's work a simple example using our native threads. Let us create an array of zeros:

Let us operate on this array simultaneously using 4 threads. We'll have each thread write its thread ID into each location.

Julia supports parallel loops using the macro. This macro is affixed in front of a loop to indicate to Julia that the loop is a multi-threaded region:

The iteration space is split amongst the threads, after which each thread writes its thread ID to its assigned locations:

Note that does not have an optional reduction parameter like .

26.16 @threadcall (Experimental)

All I/O tasks, timers, REPL commands, etc are multiplexed onto a single OS thread via an event loop. A patched version of libuv (http://docs.libuv.org/en/v1.x/) provides this functionality. Yield points provide for co-operatively scheduling multiple tasks onto the same OS thread. I/O tasks and timers yield implicitly while waiting for the event to occur. Calling explicitly allows for other tasks to be scheduled.

Thus, a task executing a effectively prevents the Julia scheduler from executing any other tasks till the call returns. This is true for all calls into external libraries. Exceptions are calls into custom C code that call back into Julia (which may then yield) or C code that calls (C equivalent of).

Note that while Julia code runs on a single thread (by default), libraries used by Julia may launch their own internal threads. For example, the BLAS library may start as many threads as there are cores on a machine.

The macro addresses scenarios where we do not want a to block the main Julia event loop. It schedules a C function for execution in a separate thread. A threadpool with a default size of 4 is used for this. The size of the threadpool is controlled via environment variable . While waiting for a free thread, and during function execution once a thread is available, the requesting task (on the main Julia event loop) yields to other tasks. Note that does not return till the execution is complete. From a user point of view, it is therefore a blocking call like other Julia APIs.

It is very important that the called function does not call back into Julia.

 may be removed/changed in future versions of Julia.

[1] In this context, MPI refers to the MPI-1 standard. Beginning with MPI-2, the MPI standards committee introduced a new set of communication mechanisms, collectively referred to as Remote Memory Access (RMA). The motivation for adding RMA to the MPI standard was to facilitate one-sided communication patterns. For additional information on the latest MPI standard, see http://mpi-forum.org/docs.

Chapter 27

Date and DateTime

The module provides two types for working with dates: and , representing day and millisecond precision, respectively; both are subtypes of the abstract . The motivation for distinct types is simple: some operations are much simpler, both in terms of code and mental reasoning, when the complexities of greater precision don't have to be dealt with. For example, since the type only resolves to the precision of a single date (i.e. no hours, minutes, or seconds), normal considerations for time zones, daylight savings/summer time, and leap seconds are unnecessary and avoided.

Both and are basically immutable wrappers. The single field of either type is actually a type, which represents a continuously increasing machine timeline based on the UT second [1]. The type is not aware of time zones (*naive*, in Python parlance), analogous to a *LocalDateTime* in Java 8. Additional time zone functionality can be added through the TimeZones.jl package, which compiles the IANA time zone database. Both and are based on the ISO 8601 standard, which follows the proleptic Gregorian calendar. One note is that the ISO 8601 standard is particular about BC/BCE dates. In general, the last day of the BC/BCE era, 1-12-31 BC/BCE, was followed by 1-1-1 AD/CE, thus no year zero exists. The ISO standard, however, states that 1 BC/BCE is year zero, so is the day before , and year (yes, negative one for the year) is 2 BC/BCE, year is 3 BC/BCE, etc.

27.1 Constructors

 and types can be constructed by integer or types, by parsing, or through adjusters (more on those later):

[1] The notion of the UT second is actually quite fundamental. There are basically two different notions of time generally accepted, one based on the physical rotation of the earth (one full rotation = 1 day), the other based on the SI second (a fixed, constant value). These are radically different! Think about it, a "UT second", as defined relative to the rotation of the earth, may have a different absolute length depending on the day! Anyway, the fact that and are based on UT seconds is a simplifying, yet honest assumption so that things like leap seconds and all their complexity can be avoided. This basis of time is formally called UT or UT1. Basing types on the UT second basically means that every minute has 60 seconds and every day has 24 hours and leads to more natural calculations when working with calendar dates.

or parsing is accomplished by the use of format strings. Format strings work by the notion of defining *delimited* or *fixed-width* "slots" that contain a period to parse and passing the text to parse and format string to a or constructor, of the form or .

Delimited slots are marked by specifying the delimiter the parser should expect between two subsequent periods; so lets the parser know that between the first and second slots in a date string like , it should find the character. The , , and characters let the parser know which periods to parse in each slot.

Fixed-width slots are specified by repeating the period character the number of times corresponding to the width with no delimiter between characters. So would correspond to a date string like . The parser distinguishes a fixed-width slot by the absence of a delimiter, noting the transition from one period character to the next.

Support for text-form month parsing is also supported through the and characters, for abbreviated and full-length month names, respectively. By default, only English month names are supported, so corresponds to "Jan", "Feb", "Mar", etc. And corresponds to "January", "February", "March", etc. Similar to other name=>value mapping functions and , custom locales can be loaded by passing in the mapping to the and dicts for abbreviated and full-name month names, respectively.

One note on parsing performance: using the function is fine if only called a few times. If there are many similarly formatted date strings to parse however, it is much more efficient to first create a , and pass it instead of a raw format string.

You can also use the string macro. This macro creates the object once when the macro is expanded and uses the same object even if a code snippet is run multiple times.

A full suite of parsing and formatting tests and examples is available in .

27.2 Durations/Comparisons

Finding the length of time between two or is straightforward given their underlying representation as and , respectively. The difference between is returned in the number of , and in the number of . Similarly, comparing is a simple matter of comparing the underlying machine instants (which in turn compares the internal values).

27.3 Accessor Functions

Because the and types are stored as single values, date parts or fields can be retrieved through accessor functions. The lowercase accessors return the field as an integer:

While propercase return the same value in the corresponding type:

Compound methods are provided, as they provide a measure of efficiency if multiple fields are needed at the same time:

One may also access the underlying or integer value:

27.4 Query Functions

Query functions provide calendrical information about a . They include information about the day of the week:

Month of the year:

As well as information about the 's year and quarter:

The and methods can also take an optional keyword that can be used to return the name of the day or month of the year for other languages/locales. There are also versions of these functions returning the abbreviated names, namely and . First the mapping is loaded into the variable:

The above mentioned functions can then be used to perform the queries:

Since the abbreviated versions of the days are not loaded, trying to use the function will error.

27.5 TimeType-Period Arithmetic

It's good practice when using any language/date framework to be familiar with how date-period arithmetic is handled as there are some tricky issues to deal with (though much less so for day-precision types).

The module approach tries to follow the simple principle of trying to change as little as possible when doing arithmetic. This approach is also often known as *calendrical* arithmetic or what you would probably guess if someone were to ask you the same calculation in a conversation. Why all the fuss about this? Let's take a classic example: add 1 month to January 31st, 2014. What's the answer? Javascript will say March 3 (assumes 31 days). PHP says March 2 (assumes 30 days). The fact is, there is no right answer. In the module, it gives the result of February 28th. How does it figure that out? I like to think of the classic 7-7-7 gambling game in casinos.

Now just imagine that instead of 7-7-7, the slots are Year-Month-Day, or in our example, 2014-01-31. When you ask to add 1 month to this date, the month slot is incremented, so now we have 2014-02-31. Then the day number is checked if it is greater than the last valid day of the new month; if it is (as in the case above), the day number is adjusted down to the last valid day (28). What are the ramifications with this approach? Go ahead and add another month to our date, . What? Were you expecting the last day of March? Nope, sorry, remember the 7-7-7 slots. As few slots as possible are going to change, so we first increment the month slot by 1, 2014-03-28, and boom, we're done because that's a valid date. On the other hand, if we were to add 2 months to our original date, 2014-01-31, then we end up with 2014-03-31, as expected. The other ramification of this approach is a loss in associativity when a specific ordering is forced (i.e. adding things in different orders results in different outcomes). For example:

What's going on there? In the first line, we're adding 1 day to January 29th, which results in 2014-01-30; then we add 1 month, so we get 2014-02-30, which then adjusts down to 2014-02-28. In the second example, we add 1 month *first*, where we get 2014-02-29, which adjusts down to 2014-02-28, and *then* add 1 day, which results in 2014-03-01. One design principle that helps in this case is that, in the presence of multiple Periods, the operations will be ordered by the Periods' *types*, not their value or positional order; this means will always be added first, then , then , etc. Hence the following *does* result in associativity and Just Works:

Tricky? Perhaps. What is an innocent user to do? The bottom line is to be aware that explicitly forcing a certain associativity, when dealing with months, may lead to some unexpected results, but otherwise, everything should work as expected. Thankfully, that's pretty much the extent of the odd cases in date-period arithmetic when dealing with time in UT (avoiding the "joys" of dealing with daylight savings, leap seconds, etc.).

As a bonus, all period arithmetic objects work directly with ranges:

27.6 Adjuster Functions

As convenient as date-period arithmetics are, often the kinds of calculations needed on dates take on a *calendrical* or *temporal* nature rather than a fixed number of periods. Holidays are a perfect example; most follow rules such as "Memorial Day = Last Monday of May", or "Thanksgiving = 4th Thursday of November". These kinds of temporal expressions

deal with rules relative to the calendar, like first or last of the month, next Tuesday, or the first and third Wednesdays, etc.

The module provides the *adjuster* API through several convenient methods that aid in simply and succinctly expressing temporal rules. The first group of adjuster methods deal with the first and last of weeks, months, quarters, and years. They each take a single as input and return or *adjust to* the first or last of the desired period relative to the input.

The next two higher-order methods, , and , generalize working with temporal expressions by taking a as first argument, along with a starting . A is just a function, usually anonymous, that takes a single as input and returns a , indicating a satisfied adjustment criterion. For example:

This is useful with the do-block syntax for more complex temporal expressions:

The method can be used to obtain all valid dates/moments in a specified range:

Additional examples and tests are available in .

27.7 Period Types

Periods are a human view of discrete, sometimes irregular durations of time. Consider 1 month; it could represent, in days, a value of 28, 29, 30, or 31 depending on the year and month context. Or a year could represent 365 or 366 days in the case of a leap year. types are simple wrappers and are constructed by wrapping any convertible type, i.e. or . Arithmetic between of the same type behave like integers, and limited arithmetic is available.

27.8 Rounding

and values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with , , or :

Unlike the numeric method, which breaks ties toward the even number by default, the method uses the rounding mode. (It's difficult to guess what breaking ties to nearest "even" would entail.) Further details on the available s can be found in the API reference.

Rounding should generally behave as expected, but there are a few cases in which the expected behaviour is not obvious.

Rounding Epoch

In many cases, the resolution specified for rounding (e.g.,) divides evenly into the next largest period (in this case,). But rounding behaviour in cases in which this is not true may lead to confusion. What is the expected result of rounding a to the nearest 10 hours?

That may seem confusing, given that the hour (12) is not divisible by 10. The reason that was chosen is that it is 17,676,660 hours after , and 17,676,660 is divisible by 10.

As Julia and values are represented according to the ISO 8601 standard, was chosen as base (or "rounding epoch") from which to begin the count of days (and milliseconds) used in rounding calculations. (Note that this differs slightly from Julia's internal representation of s using Rata Die notation; but since the ISO 8601 standard is most visible to the end user, was chosen as the rounding epoch instead of the used internally to minimize confusion.)

The only exception to the use of as the rounding epoch is when rounding to weeks. Rounding to the nearest week will always return a Monday (the first day of the week as specified by ISO 8601). For this reason, we use (the first day of the first week of year 0000, as defined by ISO 8601) as the base when rounding to a number of weeks.

Here is a related case in which the expected behaviour is not necessarily obvious: What happens when we round to the nearest , where is a type? In some cases (specifically, when) the answer is clear:

This seems obvious, because two of each of these periods still divides evenly into the next larger order period. But in the case of two months (which still divides evenly into one year), the answer may be surprising:

Why round to the first day in July, even though it is month 7 (an odd number)? The key is that months are 1-indexed (the first month is assigned 1), unlike hours, minutes, seconds, and milliseconds (the first of which are assigned 0).

This means that rounding a to an even multiple of seconds, minutes, hours, or years (because the ISO 8601 specification includes a year zero) will result in a with an even value in that field, while rounding a to an even multiple of months will result in the months field having an odd value. Because both months and years may contain an irregular number of days, whether rounding to an even number of days will result in an even value in the days field is uncertain.

See the API reference for additional information on methods exported from the module.

Chapter 28

Interacting With Julia

Julia comes with a full-featured interactive command-line REPL (read-eval-print loop) built into the executable. In addition to allowing quick and easy evaluation of Julia statements, it has a searchable history, tab-completion, many helpful keybindings, and dedicated help and shell modes. The REPL can be started by simply calling with no arguments or double-clicking on the executable:

To exit the interactive session, type – the control key together with the key on a blank line – or type followed by the return or enter key. The REPL greets you with a banner and a prompt.

28.1 The different prompt modes

The Julian mode

The REPL has four main modes of operation. The first and most common is the Julian prompt. It is the default mode of operation; each new line initially starts with . It is here that you can enter Julia expressions. Hitting return or enter after a complete expression has been entered will evaluate the entry and show the result of the last expression.

There are a number useful features unique to interactive work. In addition to showing the result, the REPL also binds the result to the variable . A trailing semicolon on the line can be used as a flag to suppress showing the result.

In Julia mode, the REPL supports something called *prompt pasting*. This activates when pasting text that starts with into the REPL. In that case, only expressions starting with are parsed, others are removed. This makes it is possible to paste a chunk of code that has been copied from a REPL session without having to scrub away prompts and outputs. This feature is enabled by default but can be disabled or enabled at will with . If it is enabled, you can try it out by pasting the code block above this paragraph straight into the REPL. This feature does not work on the standard Windows command prompt due to its limitation at detecting when a paste occurs.

Help mode

When the cursor is at the beginning of the line, the prompt can be changed to a help mode by typing . Julia will attempt to print help or documentation for anything entered in help mode:

Macros, types and variables can also be queried:

Help mode can be exited by pressing backspace at the beginning of the line.

Shell mode

Just as help mode is useful for quick access to documentation, another common task is to use the system shell to execute system commands. Just as entered help mode when at the beginning of the line, a semicolon () will enter the shell mode. And it can be exited by pressing backspace at the beginning of the line.

Search modes

In all of the above modes, the executed lines get saved to a history file, which can be searched. To initiate an incremental search through the previous history, type – the control key together with the key. The prompt will change to , and as you type the search query will appear in the quotes. The most recent result that matches the query will dynamically update to the right of the colon as more is typed. To find an older result using the same query, simply type again.

Just as is a reverse search, is a forward search, with the prompt . The two may be used in conjunction with each other to move through the previous or next matching results, respectively.

28.2 Key bindings

The Julia REPL makes great use of key bindings. Several control-key bindings were already introduced above (to exit, and for searching), but there are many more. In addition to the control-key, there are also meta-key bindings. These vary more by platform, but most terminals default to using alt- or option- held down with a key to send the meta-key (or can be configured to do so).

Customizing keybindings

Julia's REPL keybindings may be fully customized to a user's preferences by passing a dictionary to . The keys of this dictionary may be characters or strings. The key refers to the default action. Control plus character bindings are indicated with . Meta plus can be written . The values of the custom keymap must be (indicating that the input should be ignored) or functions that accept the signature . The function must be called before the REPL is initialized, by registering the operation with . For example, to bind the up and down arrow keys to move through history without prefix search, one could put the following code in :

Users should refer to to discover the available actions on key input.

28.3 Tab completion

In both the Julian and help modes of the REPL, one can enter the first few characters of a function or type and then press the tab key to get a list all matches:

Keybinding	Description
Program control	
	Exit (when buffer is empty)
	Interrupt or cancel
	Clear console screen
Return/Enter,	New line, executing if it is complete
meta-Return/Enter	Insert new line without executing it
or	Enter help or shell mode (when at start of a line)
,	Incremental history search, described above
Cursor movement	
Right arrow,	Move right one character
Left arrow,	Move left one character
Home,	Move to beginning of line
End,	Move to end of line
	Change to the previous or next history entry
	Change to the next history entry
Up arrow	Move up one line (or to the previous history entry)
Down arrow	Move down one line (or to the next history entry)
Page-up	Change to the previous history entry that matches the text before the cursor
Page-down	Change to the next history entry that matches the text before the cursor
	Move right one word
	Move left one word
	Change to the first history entry
	Change to the last history entry
Editing	
Backspace,	Delete the previous character
Delete,	Forward delete one character (when buffer has text)
meta-Backspace	Delete the previous word
	Forward delete the next word
	Delete previous text up to the nearest whitespace
	”Kill” to end of line, placing the text in a buffer
	”Yank” insert the text from the kill buffer
	Transpose the characters about the cursor
	Write a number in REPL and press to open editor at corresponding stackframe or method

The tab key can also be used to substitute LaTeX math symbols with their Unicode equivalents, and get a list of LaTeX matches as well:

A full list of tab-completions can be found in the Unicode Input section of the manual.

Completion of paths works for strings and julia's shell mode:

Tab completion can help with investigation of the available methods matching the input arguments:

Keywords are also displayed in the suggested methods, see second line after where and are keyword arguments:

The completion of the methods uses type inference and can therefore see if the arguments match even if the arguments are output from functions. The function needs to be type stable for the completion to be able to remove non-matching methods.

Tab completion can also help completing fields:

Fields for output from functions can also be completed:

The completion of fields for output from functions uses type inference, and it can only suggest fields if the function is type stable.

28.4 Customizing Colors

The colors used by Julia and the REPL can be customized, as well. To change the color of the Julia prompt you can add something like the following to your file, which is to be placed inside your home directory:

The available color keys can be seen by typing in the help mode of the REPL. In addition, the integers 0 to 255 can be used as color keys for terminals with 256 color support.

You can also change the colors for the help and shell prompts and input and answer text by setting the appropriate field of in the function above (respectively, , , , and). For the latter two, be sure that the field is also set to false.

It is also possible to apply boldface formatting by using as a color. For instance, to print answers in boldface font, one can use the following as a :

You can also customize the color used to render warning and informational messages by setting the appropriate environment variables. For instance, to render error, warning, and informational messages respectively in magenta, yellow, and cyan you can add the following to your file:

Chapter 29

Running External Programs

Julia borrows backtick notation for commands from the shell, Perl, and Ruby. However, in Julia, writing

differs in several aspects from the behavior in various shells, Perl, or Ruby:

- Instead of immediately running the command, backticks create a object to represent the command. You can use this object to connect the command to others via pipes, run it, and read or write to it.
- When the command is run, Julia does not capture its output unless you specifically arrange for it to. Instead, the output of the command by default goes to as it would using 's call.
- The command is never run with a shell. Instead, Julia parses the command syntax directly, appropriately interpolating variables and splitting on words as the shell would, respecting shell quoting syntax. The command is run as 's immediate child process, using and calls.

Here's a simple example of running an external program:

The is the output of the command, sent to . The run method itself returns , and throws an if the external command fails to run successfully.

If you want to read the output of the external command, can be used instead:

More generally, you can use to read from or write to an external command.

The program name and the individual arguments in a command can be accessed and iterated over as if the command were an array of strings:

29.1 Interpolation

Suppose you want to do something a bit more complicated and use the name of a file in the variable as an argument to a command. You can use for interpolation much as you would in a string literal (see Strings):

A common pitfall when running external programs via a shell is that if a file name contains characters that are special to the shell, they may cause undesirable behavior. Suppose, for example, rather than , we wanted to sort the contents of the file . Let's try it:

How did the file name get quoted? Julia knows that is meant to be interpolated as a single argument, so it quotes the word for you. Actually, that is not quite accurate: the value of is never interpreted by a shell, so there's no need for actual quoting; the quotes are inserted only for presentation to the user. This will even work if you interpolate a value as part of a shell word:

As you can see, the space in the variable is appropriately escaped. But what if you *want* to interpolate multiple words? In that case, just use an array (or any other iterable container):

If you interpolate an array as part of a shell word, Julia emulates the shell's argument generation:

Moreover, if you interpolate multiple arrays into the same word, the shell's Cartesian product generation behavior is emulated:

Since you can interpolate literal arrays, you can use this generative functionality without needing to create temporary array objects first:

29.2 Quoting

Inevitably, one wants to write commands that aren't quite so simple, and it becomes necessary to use quotes. Here's a simple example of a Perl one-liner at a shell prompt:

The Perl expression needs to be in single quotes for two reasons: so that spaces don't break the expression into multiple shell words, and so that uses of Perl variables like (yes, that's the name of a variable in Perl), don't cause interpolation. In other instances, you may want to use double quotes so that interpolation *does* occur:

In general, the Julia backtick syntax is carefully designed so that you can just cut-and-paste shell commands as is into backticks and they will work: the escaping, quoting, and interpolation behaviors are the same as the shell's. The only difference is that the interpolation is integrated and aware of Julia's notion of what is a single string value, and what is a container for multiple values. Let's try the above two examples in Julia:

The results are identical, and Julia's interpolation behavior mimics the shell's with some improvements due to the fact that Julia supports first-class iterable objects while most shells use strings split on spaces for this, which introduces ambiguities. When trying to port shell commands to Julia, try cut and pasting first. Since Julia shows commands to you before running them, you can easily and safely just examine its interpretation without doing any damage.

29.3 Pipelines

Shell metacharacters, such as , , and , need to be quoted (or escaped) inside of Julia's backticks:

This expression invokes the command with three words as arguments: , , and . The result is that a single line is printed: . How, then, does one construct a pipeline? Instead of using inside of backticks, one uses :

This pipes the output of the command to the command. Of course, this isn't terribly interesting since there's only one line to sort, but we can certainly do much more interesting things:

This prints the highest five user IDs on a UNIX system. The , and commands are all spawned as immediate children of the current process, with no intervening shell process. Julia itself does the work to setup pipes and connect file descriptors that is normally done by the shell. Since Julia does this itself, it retains better control and can do some things that shells cannot.

Julia can run multiple commands in parallel:

The order of the output here is non-deterministic because the two processes are started nearly simultaneously, and race to make the first write to the descriptor they share with each other and the parent process. Julia lets you pipe the output from both of these processes to another program:

In terms of UNIX plumbing, what's happening here is that a single UNIX pipe object is created and written to by both processes, and the other end of the pipe is read from by the command.

IO redirection can be accomplished by passing keyword arguments stdin, stdout, and stderr to the function:

Avoiding Deadlock in Pipelines

When reading and writing to both ends of a pipeline from a single process, it is important to avoid forcing the kernel to buffer all of the data.

For example, when reading all of the output from a command, call , not , since the former will actively consume all of the data written by the process, whereas the latter will attempt to store the data in the kernel's buffers while waiting for a reader to be connected.

Another common solution is to separate the reader and writer of the pipeline into separate Tasks:

Complex Example

The combination of a high-level programming language, a first-class command abstraction, and automatic setup of pipes between processes is a powerful one. To give some sense of the complex pipelines that can be created easily, here are some more sophisticated examples, with apologies for the excessive use of Perl one-liners:

This is a classic example of a single producer feeding two concurrent consumers: one process generates lines with the numbers 0 through 9 on them, while two parallel processes consume that output, one prefixing lines with the letter "A", the other with the letter "B". Which consumer gets the first line is non-deterministic, but once that race has been won, the lines are consumed alternately by one process and then the other. (Setting in Perl causes each print statement to flush the handle, which is necessary for this example to work. Otherwise all the output is buffered and printed to the pipe at once, to be read by just one consumer process.)

Here is an even more complex multi-stage producer-consumer example:

This example is similar to the previous one, except there are two stages of consumers, and the stages have different latency so they use a different number of parallel workers, to maintain saturated throughput.

We strongly encourage you to try all these examples to see how they work.

Chapter 30

Calling C and Fortran Code

Though most code can be written in Julia, there are many high-quality, mature libraries for numerical computing already written in C and Fortran. To allow easy use of this existing code, Julia makes it simple and efficient to call C and Fortran functions. Julia has a "no boilerplate" philosophy: functions can be called directly from Julia without any "glue" code, code generation, or compilation – even from the interactive prompt. This is accomplished just by making an appropriate call with syntax, which looks like an ordinary function call.

The code to be called must be available as a shared library. Most C and Fortran libraries ship compiled as shared libraries already, but if you are compiling the code yourself using GCC (or Clang), you will need to use the and options. The machine instructions generated by Julia's JIT are the same as a native C call would be, so the resulting overhead is the same as calling a library function from C code. (Non-library function calls in both C and Julia can be inlined and thus may have even less overhead than calls to shared library functions. When both libraries and executables are generated by LLVM, it is possible to perform whole-program optimizations that can even optimize across this boundary, but Julia does not yet support that. In the future, however, it may do so, yielding even greater performance gains.)

Shared libraries and functions are referenced by a tuple of the form or where is the C-exported function name. refers to the shared library name: shared libraries available in the (platform-specific) load path will be resolved by name, and if necessary a direct path may be specified.

A function name may be used alone in place of the tuple (just or). In this case the name is resolved within the current process. This form can be used to call C library functions, functions in the Julia runtime, or functions in an application linked to Julia.

By default, Fortran compilers generate mangled names (for example, converting function names to lowercase or uppercase, often appending an underscore), and so to call a Fortran function via you must pass the mangled identifier corresponding to the rule followed by your Fortran compiler. Also, when calling a Fortran function, all inputs must be passed by reference.

Finally, you can use to actually generate a call to the library function. Arguments to are as follows:

1. A pair, which must be written as a literal constant,

   OR

   a function pointer (for example, from).

2. Return type (see below for mapping the declared C type to Julia)

   - This argument will be evaluated at compile-time, when the containing method is defined.

3. A tuple of input types. The input types must be written as a literal tuple, not a tuple-valued variable or expression.

   - This argument will be evaluated at compile-time, when the containing method is defined.

4. The following arguments, if any, are the actual argument values passed to the function.

As a complete but simple example, the following calls the function from the standard C library:

takes no arguments and returns an . One common gotcha is that a 1-tuple must be written with a trailing comma. For example, to call the function to get a pointer to the value of an environment variable, one makes a call like this:

Note that the argument type tuple must be written as , rather than . This is because is just the expression surrounded by parentheses, rather than a 1-tuple containing :

In practice, especially when providing reusable functionality, one generally wraps uses in Julia functions that set up arguments and then check for errors in whatever manner the C or Fortran function indicates them, propagating to the Julia caller as exceptions. This is especially important since C and Fortran APIs are notoriously inconsistent about how they indicate error conditions. For example, the C library function is wrapped in the following Julia function, which is a simplified version of the actual definition from :

The C function indicates an error by returning , but other standard C functions indicate errors in various different ways, including by returning -1, 0, 1 and other special values. This wrapper throws an exception clearly indicating the problem if the caller tries to get a non-existent environment variable:

Here is a slightly more complex example that discovers the local machine's hostname:

This example first allocates an array of bytes, then calls the C library function to fill the array in with the hostname, takes a pointer to the hostname buffer, and converts the pointer to a Julia string, assuming that it is a NUL-terminated C string. It is common for C libraries to use this pattern of requiring the caller to allocate memory to be passed to the callee and filled in. Allocation of memory from Julia like this is generally accomplished by creating an uninitialized array and passing a pointer to its data to the C function. This is why we don't use the type here: as the array is uninitialized, it could contain NUL bytes. Converting to a as part of the checks for contained NUL bytes and could therefore throw a conversion error.

30.1 Creating C-Compatible Julia Function Pointers

It is possible to pass Julia functions to native C functions that accept function pointer arguments. For example, to match C prototypes of the form:

The function generates the C-compatible function pointer for a call to a Julia function. Arguments to are as follows:

1. A Julia Function
2. Return type
3. A tuple of input types

Only platform-default C calling convention is supported. -generated pointers cannot be used in calls where WINAPI expects function on 32-bit windows, but can be used on WIN64 (where is unified with C calling convention).

A classic example is the standard C library function, declared as:

The argument is a pointer to an array of length , with elements of bytes each. is a callback function which takes pointers to two elements and and returns an integer less/greater than zero if should appear before/after (or zero if any order is permitted). Now, suppose that we have a 1d array of values in Julia that we want to sort using the function (rather than Julia's built-in function). Before we worry about calling and passing arguments, we need to write a comparison function that works for some arbitrary type T:

Notice that we have to be careful about the return type: expects a function returning a C , so we must be sure to return via a call to and a .

In order to pass this function to C, we obtain its address using the function :

accepts three arguments: the Julia function (), the return type (), and a tuple of the argument types, in this case to sort an array of () elements.

The final call to looks like this:

As can be seen, is changed to the sorted array . Note that Julia knows how to convert an array into a , how to compute the size of a type in bytes (identical to C's operator), and so on. For fun, try inserting a line into , which will allow you to see the comparisons that is performing (and to verify that it is really calling the Julia function that you passed to it).

30.2 Mapping C Types to Julia

It is critical to exactly match the declared C type with its declaration in Julia. Inconsistencies can cause code that works correctly on one system to fail or produce indeterminate results on a different system.

Note that no C header files are used anywhere in the process of calling C functions: you are responsible for making sure that your Julia types and call signatures accurately reflect those in the C header file. (The Clang package can be used to auto-generate Julia code from a C header file.)

Auto-conversion:

Julia automatically inserts calls to the function to convert each argument to the specified type. For example, the following call:

will behave as if the following were written:

normally just calls , but can be defined to return an arbitrary new object more appropriate for passing to C. This should be used to perform all allocations of memory that will be accessed by the C code. For example, this is used to convert an of objects (e.g. strings) to an array of pointers.

handles conversion to types. It is considered unsafe because converting an object to a native pointer can hide the object from the garbage collector, causing it to be freed prematurely.

Type Correspondences:

First, a review of some relevant Julia type terminology:

Syntax / Keyword	Example	Description
		”Leaf Type” :: A group of related data that includes a type-tag, is managed by the Julia GC, and is defined by object-identity. The type parameters of a leaf type must be fully defined (no are allowed) in order for the instance to be constructed.
	, ,	”Super Type” :: A super-type (not a leaf-type) that cannot be instantiated, but can be used to describe a group of types.
		”Type Parameter” :: A specialization of a type (typically used for dispatch or storage optimization).
		”TypeVar” :: The in the type parameter declaration is referred to as a TypeVar (short for type variable).
	,	”Primitive Type” :: A type with no fields, but a size. It is stored and defined by-value.
		”Struct” :: A type with all fields defined to be constant. It is defined by-value, and may be stored with a type-tag.
	()	”Is-Bits” :: A , or a type where all fields are other types. It is defined by-value, and is stored without a type-tag.
		”Singleton” :: a Leaf Type or Struct with no fields.
or		”Tuple” :: an immutable data-structure similar to an anonymous struct type, or a constant array. Represented as either an array or a struct.

Bits Types:

There are several special types to be aware of, as no other type can be defined to behave the same:

- Exactly corresponds to the type in C (or in Fortran).
- Exactly corresponds to the type in C (or in Fortran).
- Exactly corresponds to the type in C (or in Fortran).

- Exactly corresponds to the type in C (or in Fortran).

- Exactly corresponds to the type annotation in C (or any type in Fortran). Any Julia type that is not a subtype of is assumed to be unsigned.

- Behaves like a that can manage its memory via the Julia GC.

- When an array is passed to C as a argument, it is not reinterpret-cast: Julia requires that the element type of the array matches , and the address of the first element is passed.

  Therefore, if an contains data in the wrong format, it will have to be explicitly converted using a call such as .

  To pass an array as a pointer of a different type *without* converting the data beforehand (for example, to pass a array to a function that operates on uninterpreted bytes), you can declare the argument as .

  If an array of eltype is passed as a argument, will attempt to first make a null-terminated copy of the array with each element replaced by its version. This allows, for example, passing an pointer array of type to an argument of type .

On all systems we currently support, basic C/C++ value types may be translated to Julia types as follows. Every C type also has a corresponding Julia type with the same name, prefixed by C. This can help for writing portable code (and remembering that an in C is not the same as an in Julia).

System Independent:

The type is essentially a synonym for , except the conversion to throws an error if the Julia string contains any embedded NUL characters (which would cause the string to be silently truncated if the C routine treats NUL as the terminator). If you are passing a to a C routine that does not assume NUL termination (e.g. because you pass an explicit string length), or if you know for certain that your Julia string does not contain NUL and want to skip the check, you can use as the argument type. can also be used as the return type, but in that case it obviously does not introduce any extra checks and is only meant to improve readability of the call.

System-dependent:

> **Note**
>
> When calling a Fortran function, all inputs must be passed by reference, so all type correspondences above should contain an additional or wrapper around their type specification.

> **Warning**
>
> For string arguments () the Julia type should be (if NUL- terminated data is expected) or either or otherwise (these two pointer types have the same effect), as described above, not . Similarly, for array arguments (or), the Julia type should again be , not .

> **Warning**
>
> Julia's type is 32 bits, which is not the same as the wide character type (or) on all platforms.

C name	Fortran name	Standard Julia Alias	Julia Base Type
(only in C++)			
	,		
, (C, typical)	,		
	,		
and or			
(where T represents an appropriately defined type)			
(or , e.g. a string)			if NUL-terminated, or if not
(or)			
(any Julia Type)			
(a reference to a Julia Type)			
			Not supported
(variadic function specification)			(where is one of the above types, variadic functions of different argument types are not supported)

C name	Standard Julia Alias	Julia Base Type
		(x86, x86_64), (powerpc, arm)
		(UNIX), (Windows)
		(UNIX), (Windows)
		(UNIX), (Windows)

Warning

A return type of means the function will not return i.e. C++11 or C11 (e.g. or). Do not use this for functions that return no value () but do return, use instead.

Note

For arguments, the Julia type should be (if the C routine expects a NUL-terminated string) or otherwise. Note also that UTF-8 string data in Julia is internally NUL-terminated, so it can be passed to C functions expecting NUL-terminated data without making a copy (but using the type will cause an error to be thrown if the string itself contains NUL characters).

Note

C functions that take an argument of the type can be called by using a type within Julia. For example, C functions of the form:

can be called via the following Julia code:

Note

A C function declared to return will return the value in Julia.

Struct Type correspondences

Composite types, aka in C or in Fortran90 (or / in some variants of F77), can be mirrored in Julia by creating a definition with the same field layout.

When used recursively, types are stored inline. All other types are stored as a pointer to the data. When mirroring a struct used by-value inside another struct in C, it is imperative that you do not attempt to manually copy the fields over, as this will not preserve the correct field alignment. Instead, declare an struct type and use that instead. Unnamed structs are not possible in the translation to Julia.

Packed structs and union declarations are not supported by Julia.

You can get a near approximation of a if you know, a priori, the field that will have the greatest size (potentially including padding). When translating your fields to Julia, declare the Julia field to be only of that type.

Arrays of parameters can be expressed with :

Arrays of unknown size (C99-compliant variable length structs specified by or) are not directly supported. Often the best way to deal with these is to deal with the byte offsets directly. For example, if a C library declared a proper string type and returned a pointer to it:

In Julia, we can access the parts independently to make a copy of that string:

Type Parameters

The type arguments to are evaluated statically, when the method containing the ccall is defined. They therefore must take the form of a literal tuple, not a variable, and cannot reference local variables.

This may sound like a strange restriction, but remember that since C is not a dynamic language like Julia, its functions can only accept argument types with a statically-known, fixed signature.

However, while the type layout must be known statically to compute the ABI, the static parameters of the function are considered to be part of this static environment. The static parameters of the function may be used as type parameters in the signature, as long as they don't affect the layout of the type. For example, is valid, since is always a word-size primitive type. But, is not valid, since the type layout of is not known statically.

SIMD Values

Note: This feature is currently implemented on 64-bit x86 and AArch64 platforms only.

If a C/C++ routine has an argument or return value that is a native SIMD type, the corresponding Julia type is a homogeneous tuple of that naturally maps to the SIMD type. Specifically:

- The tuple must be the same size as the SIMD type. For example, a tuple representing an on x86 must have a size of 16 bytes.
- The element type of the tuple must be an instance of where is a primitive type that is 1, 2, 4 or 8 bytes.

For instance, consider this C routine that uses AVX intrinsics:

The following Julia code calls using :

The host machine must have the requisite SIMD registers. For example, the code above will not work on hosts without AVX support.

Memory Ownership

malloc/free

Memory allocation and deallocation of such objects must be handled by calls to the appropriate cleanup routines in the libraries being used, just like in any C program. Do not try to free an object received from a C library with in Julia, as this may result in the function being called via the wrong library and cause Julia to crash. The reverse (passing an object allocated in Julia to be freed by an external library) is equally invalid.

When to use T, Ptr{T} and Ref{T}

In Julia code wrapping calls to external C routines, ordinary (non-pointer) data should be declared to be of type inside the , as they are passed by value. For C code accepting pointers, should generally be used for the types of input arguments, allowing the use of pointers to memory managed by either Julia or C through the implicit call to . In contrast, pointers returned by the C function called should be declared to be of output type , reflecting that the memory pointed to is managed by C only. Pointers contained in C structs should be represented as fields of type within the corresponding Julia struct types designed to mimic the internal structure of corresponding C structs.

In Julia code wrapping calls to external Fortran routines, all input arguments should be declared as of type , as Fortran passes all variables by reference. The return type should either be for Fortran subroutines, or a for Fortran functions returning the type .

30.3 Mapping C Functions to Julia

/ argument translation guide

For translating a C argument list to Julia:

- , where is one of the primitive types: , , , , , , , or any of their equivalents
  - , where is an equivalent Julia Bits Type (per the table above)
  - if is an , the argument type should be equivalent to or
  - argument value will be copied (passed by value)
- (including typedef to a struct)
  - , where is a Julia leaf type
  - argument value will be copied (passed by value)
-
  - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
  - this argument may be declared as , if it really is just an unknown pointer
-
  -
  - argument value must be a valid Julia object
  - currently unsupported by
-
  -
  - argument value must be a valid Julia object (or)

  - currently unsupported by

-
  - , where is the Julia type corresponding to
  - argument value will be copied if it is an type otherwise, the value must be a valid Julia object
- (e.g. a pointer to a function)
  - (you may need to use explicitly to create this pointer)
- (e.g. a vararg)
  - , where is the Julia type
-
  - not supported

/ return type translation guide

For translating a C return type to Julia:

-
  - (this will return the singleton instance)
- , where is one of the primitive types: , , , , , , , or any of their equivalents
  - , where is an equivalent Julia Bits Type (per the table above)
  - if is an , the argument type should be equivalent to or
  - argument value will be copied (returned by-value)
- (including typedef to a struct)
  - , where is a Julia Leaf Type
  - argument value will be copied (returned by-value)
-
  - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
  - this argument may be declared as , if it really is just an unknown pointer
-
  -
  - argument value must be a valid Julia object
-
  -
  - argument value must be a valid Julia object (or)

-
  - If the memory is already owned by Julia, or is an type, and is known to be non-null:
    - , where is the Julia type corresponding to
    - a return type of is invalid, it should either be (corresponding to) or (corresponding to)
    - C **MUST NOT** modify the memory returned via if is an type
  - If the memory is owned by C:
    - , where is the Julia type corresponding to
- (e.g. a pointer to a function)
  - (you may need to use explicitly to create this pointer)

Passing Pointers for Modifying Inputs

Because C doesn't support multiple return values, often C functions will take pointers to data that the function will modify. To accomplish this within a , you need to first encapsulate the value inside an of the appropriate type. When you pass this object as an argument, Julia will automatically pass a C pointer to the encapsulated data:

Upon return, the contents of and can be retrieved (if they were changed by) by and ; that is, they act like zero-dimensional arrays.

Special Reference Syntax for ccall (deprecated):

The syntax is deprecated, use the argument type instead.

A prefix is used on an argument to to indicate that a pointer to a scalar argument should be passed instead of the scalar value itself (required for all Fortran function arguments, as noted above). The following example computes a dot product using a BLAS function.

The meaning of prefix is not quite the same as in C. In particular, any changes to the referenced variables will not be visible in Julia unless the type is mutable (declared via). However, even for immutable structs it will not cause any harm for called functions to attempt such modifications (that is, writing through the passed pointers). Moreover, may be used with any expression, such as or .

When a scalar value is passed with as an argument of type , the value will first be converted to type .

30.4 Some Examples of C Wrappers

Here is a simple example of a C wrapper that returns a type:

The GNU Scientific Library (here assumed to be accessible through) defines an opaque pointer, , as the return type of the C function . As user code never has to look inside the struct, the corresponding Julia wrapper simply needs a new type declaration, , that has no internal fields and whose sole purpose is to be placed in the type parameter of a type. The return type of the is declared as , since the memory allocated and pointed to by is controlled by C (and not Julia).

The input is passed by value, and so the function's input signature is simply declared as without any or necessary. (If the wrapper was calling a Fortran function instead, the corresponding function input signature should instead be , since Fortran variables are passed by reference.) Furthermore, can be any type that is convertable to a integer; the implicitly calls .

Here is a second example wrapping the corresponding destructor:

Here, the input is declared to be of type , meaning that the memory that points to may be managed by Julia or by C. A pointer to memory allocated by C should be of type , but it is convertable using and therefore can be used in the same (covariant) context of the input argument to a . A pointer to memory allocated by Julia must be of type , to ensure that the memory address pointed to is valid and that Julia's garbage collector manages the chunk of memory pointed to correctly. Therefore, the declaration allows pointers managed by C or Julia to be used.

If the C wrapper never expects the user to pass pointers to memory managed by Julia, then using for the method signature of the wrapper and similarly in the is also acceptable.

Here is a third example passing Julia arrays:

The C function wrapped returns an integer error code; the results of the actual evaluation of the Bessel J function populate the Julia array . This variable can only be used with corresponding input type declaration , since its memory is allocated and managed by Julia, not C. The implicit call to unpacks the Julia pointer to a Julia array data structure into a form understandable by C.

Note that for this code to work correctly, must be declared to be of type and not . The memory is managed by Julia and the signature alerts Julia's garbage collector to keep managing the memory for while the executes. If were used instead, the may still work, but Julia's garbage collector would not be aware that the memory declared for is being used by the external C function. As a result, the code may produce a memory leak if never gets freed by the garbage collector, or if the garbage collector prematurely frees , the C function may end up throwing an invalid memory access exception.

30.5 Garbage Collection Safety

When passing data to a , it is best to avoid using the function. Instead define a convert method and pass the variables directly to the . automatically arranges that all of its arguments will be preserved from garbage collection until the call returns. If a C API will store a reference to memory allocated by Julia, after the returns, you must arrange that the object remains visible to the garbage collector. The suggested way to handle this is to make a global variable of type to hold these values, until the C library notifies you that it is finished with them.

Whenever you have created a pointer to Julia data, you must ensure the original data exists until you are done with using the pointer. Many methods in Julia such as and make copies of data instead of taking ownership of the buffer, so that it is safe to free (or alter) the original data without affecting Julia. A notable exception is which, for performance reasons, shares (or can be told to take ownership of) the underlying buffer.

The garbage collector does not guarantee any order of finalization. That is, if contained a reference to and both and are due for garbage collection, there is no guarantee that would be finalized after . If proper finalization of depends on being valid, it must be handled in other ways.

30.6 Non-constant Function Specifications

A function specification must be a constant expression. However, it is possible to use computed values as function names by staging through as follows:

This expression constructs a name using , then substitutes this name into a new expression, which is then evaluated. Keep in mind that only operates at the top level, so within this expression local variables will not be available (unless their values are substituted with). For this reason, is typically only used to form top-level definitions, for example when wrapping libraries that contain many similar functions.

If your usage is more dynamic, use indirect calls as described in the next section.

30.7 Indirect Calls

The first argument to can also be an expression evaluated at run time. In this case, the expression must evaluate to a , which will be used as the address of the native function to call. This behavior occurs when the first argument contains references to non-constants, such as local variables, function arguments, or non-constant globals.

For example, you might look up the function via , then cache it in a shared reference for that session. For example:

30.8 Calling Convention

The second argument to can optionally be a calling convention specifier (immediately preceding return type). Without any specifier, the platform-default C calling convention is used. Other supported conventions are: , , , and (no-op on 64-bit Windows). For example (from) we see the same as above, but with the correct signature for Windows:

For more information, please see the LLVM Language Reference.

There is one additional special calling convention , which allows inserting calls to LLVM intrinsics directly. This can be especially useful when targeting unusual platforms such as GPGPUs. For example, for CUDA, we need to be able to read the thread index:

As with any , it is essential to get the argument signature exactly correct. Also, note that there is no compatibility layer that ensures the intrinsic makes sense and works on the current target, unlike the equivalent Julia functions exposed by .

30.9 Accessing Global Variables

Global variables exported by native libraries can be accessed by name using the function. The arguments to are a symbol specification identical to that used by , and a type describing the value stored in the variable:

The result is a pointer giving the address of the value. The value can be manipulated through this pointer using and .

30.10 Accessing Data through a Pointer

The following methods are described as "unsafe" because a bad pointer or type declaration can cause Julia to terminate abruptly.

Given a , the contents of type can generally be copied from the referenced memory into a Julia object using . The index argument is optional (default is 1), and follows the Julia-convention of 1-based indexing. This function is intentionally similar to the behavior of and (e.g. access syntax).

The return value will be a new object initialized to contain a copy of the contents of the referenced memory. The referenced memory can safely be freed or released.

If is , then the memory is assumed to contain a reference to a Julia object (a), the result will be a reference to this object, and the object will not be copied. You must be careful in this case to ensure that the object was always visible to the garbage collector (pointers do not count, but the new reference does) to ensure the memory is not prematurely freed. Note that if the object was not originally allocated by Julia, the new object will never be finalized by Julia's garbage collector. If the itself is actually a , it can be converted back to a Julia object reference by . (Julia values can be converted to pointers, as , by calling .)

The reverse operation (writing data to a), can be performed using . Currently, this is only supported for primitive types or other pointer-free () immutable struct types.

Any operation that throws an error is probably currently unimplemented and should be posted as a bug so that it can be resolved.

If the pointer of interest is a plain-data array (primitive type or immutable struct), the function may be more useful. The final parameter should be true if Julia should "take ownership" of the underlying buffer and call when the returned object is finalized. If the parameter is omitted or false, the caller must ensure the buffer remains in existence until all access is complete.

Arithmetic on the type in Julia (e.g. using) does not behave the same as C's pointer arithmetic. Adding an integer to a in Julia always moves the pointer by some number of *bytes*, not elements. This way, the address values obtained from pointer arithmetic do not depend on the element types of pointers.

30.11 Thread-safety

Some C libraries execute their callbacks from a different thread, and since Julia isn't thread-safe you'll need to take some extra precautions. In particular, you'll need to set up a two-layered system: the C callback should only *schedule* (via Julia's event loop) the execution of your "real" callback. To do this, create a object and wait on it:

The callback you pass to C should only execute a to , passing as the argument, taking care to avoid any allocations or other interactions with the Julia runtime.

Note that events may be coalesced, so multiple calls to may result in a single wakeup notification to the condition.

30.12 More About Callbacks

For more details on how to pass callbacks to C libraries, see this blog post.

30.13 C++

For direct C++ interfacing, see the Cxx package. For tools to create C++ bindings, see the CxxWrap package.

Chapter 31

Handling Operating System Variation

When dealing with platform libraries, it is often necessary to provide special cases for various platforms. The variable can be used to write these special cases. There are several functions in the module intended to make this easier: , , , , and . These may be used as follows:

Note that and are mutually exclusive subsets of . Additionally, there is a macro which makes it possible to use these functions to conditionally hide invalid code, as demonstrated in the following examples.

Simple blocks:

Complex blocks:

When chaining conditionals (including //), the must be repeated for each level (parentheses optional, but recommended for readability):

Chapter 32

Environment Variables

Julia may be configured with a number of environment variables, either in the usual way of the operating system, or in a portable way from within Julia. Suppose you want to set the environment variable to , then either type for instance in the REPL to make this change on a case by case basis, or add the same to the user configuration file in the user's home directory to have a permanent effect. The current value of the same environment variable is determined by evaluating .

The environment variables that Julia uses generally start with . If is called with equal to , then the output will list defined environment variables relevant for Julia, including those for which appears in the name.

32.1 File locations

The absolute path of the directory containing the Julia executable, which sets the global variable . If is not set, then Julia determines the value at run-time.

The executable itself is one of

```
```

by default.

The global variable determines a relative path from to the data directory associated with Julia. Then the path

```
```

determines the directory in which Julia initially searches for source files (via).

Likewise, the global variable determines a relative path to the configuration file directory. Then Julia searches for a file at

```
```

by default (via).

For example, a Linux installation with a Julia executable located at , a of , and a of will have set to , a source-file search path of

```
```

and a global configuration search path of

```
```

A separated list of absolute paths that are to be appended to the variable . (In Unix-like systems, the path separator is ; in Windows systems, the path separator is .) The variable is where and look for code; it defaults to the absolute paths

so that, e.g., version 0.6 of Julia on a Linux system with a Julia executable at will have a default of

The path of the parent directory for the version-specific Julia package repositories. If the path is relative, then it is taken with respect to the working directory. If is not set, then defaults to

Then the repository location for a given Julia version is

For example, for a Linux user whose home directory is , the directory containing the package repositories would by default be

and the package repository for version 0.6 of Julia would be

The absolute path of the REPL's history file. If is not set, then defaults to

A positive that determines how much time the max-sum subroutine of the package dependency resolver will devote to attempting satisfying constraints before giving up: this value is by default , and larger values correspond to larger amounts of time.

Suppose the value of is . Then

- the number of pre-decimation iterations is ,
- the number of iterations between decimation steps is , and
- at decimation steps, at most one in every packages is decimated.

32.2 External applications

The absolute path of the shell with which Julia should execute external commands (via). Defaults to the environment variable , and falls back to if is unset.

Note

On Windows, this environment variable is ignored, and external commands are executed directly.

The editor returned by and used in, e.g., , referring to the command of the preferred editor, for instance .

 takes precedence over , which in turn takes precedence over . If none of these environment variables is set, then the editor is taken to be on Windows and OS X, or if it exists, or otherwise.

Note

 is *not* used in the determination of the editor for : this function checks and alone.

32.3 Parallelization

Overrides the global variable , the number of logical CPU cores available.

A that sets the value of (default:). This function gives the number of seconds a worker process will wait for a master process to establish a connection before dying.

An unsigned 64-bit integer () that sets the maximum number of threads available to Julia. If exceeds the number of available physical CPU cores, then the number of threads is set to the number of cores. If is not positive or is not set, or if the number of CPU cores cannot be determined through system calls, then the number of threads is set to .

If set to a string that starts with the case-insensitive substring , then spinning threads never sleep. Otherwise, is interpreted as an unsigned 64-bit integer () and gives, in nanoseconds, the amount of time after which spinning threads should sleep.

If set to anything besides , then Julia's thread policy is consistent with running on a dedicated machine: the master thread is on proc 0, and threads are affinitized. Otherwise, Julia lets the operating system handle thread policy.

32.4 REPL formatting

Environment variables that determine how REPL output should be formatted at the terminal. Generally, these variables should be set to ANSI terminal escape sequences. Julia provides a high-level interface with much of the same functionality: see the section on Interacting With Julia.

The formatting (default: light red,) that errors should have at the terminal.

The formatting (default: yellow,) that warnings should have at the terminal.

The formatting (default: cyan,) that info should have at the terminal.

The formatting (default: normal,) that input should have at the terminal.

The formatting (default: normal,) that output should have at the terminal.

The formatting (default: bold,) that line info should have during a stack trace at the terminal.

The formatting (default: bold,) that function calls should have during a stack trace at the terminal.

32.5 Debugging and profiling

, ,

If set, these environment variables take strings that optionally start with the character , followed by a string interpolation of a colon-separated list of three signed 64-bit integers (). This triple of integers represents the arithmetic sequence , , , … .

- If it's the th time that has been called, and belongs to the arithmetic sequence represented by , then garbage collection is forced.
- If it's the th time that has been called, and belongs to the arithmetic sequence represented by , then garbage collection is forced.
- If it's the th time that has been called, and belongs to the arithmetic sequence represented by , then counts for the number of calls to and are printed.

If the value of the environment variable begins with the character , then the interval between garbage collection events is randomized.

> **Note**
>
> These environment variables only have an effect if Julia was compiled with garbage-collection debugging (that is, if is set to in the build configuration).

If set to anything besides , then the Julia garbage collector never performs ”quick sweeps” of memory.

Note

This environment variable only has an effect if Julia was compiled with garbage-collection debugging (that is, if is set to in the build configuration).

If set to anything besides , then the Julia garbage collector will wait for a debugger to attach instead of aborting whenever there's a critical error.

Note

This environment variable only has an effect if Julia was compiled with garbage-collection debugging (that is, if is set to in the build configuration).

If set to anything besides , then the compiler will create and register an event listener for just-in-time (JIT) profiling.

Note

This environment variable only has an effect if Julia was compiled with JIT profiling support, using either

- Intel's VTune™ Amplifier (set to in the build configuration), or
- OProfile (set to in the build configuration).

Arguments to be passed to the LLVM backend.

Note

This environment variable has an effect only if Julia was compiled with set — in particular, the executable is always compiled with this build variable.

If set, then Julia prints detailed information about the cache in the loading process of .

Chapter 33

Embedding Julia

As we have seen in Calling C and Fortran Code, Julia has a simple and efficient way to call functions written in C. But there are situations where the opposite is needed: calling Julia function from C code. This can be used to integrate Julia code into a larger C/C++ project, without the need to rewrite everything in C/C++. Julia has a C API to make this possible. As almost all programming languages have some way to call C functions, the Julia C API can also be used to build further language bridges (e.g. calling Julia from Python or C#).

33.1 High-Level Embedding

We start with a simple C program that initializes Julia and calls some Julia code:

In order to build this program you have to put the path to the Julia header into the include path and link against . For instance, when Julia is installed to , one can compile the above test program with using:

Then if the environment variable is set to , the output program can be executed.

Alternatively, look at the program in the Julia source tree in the folder. The file program is another simple example of how to set options while linking against .

The first thing that has to be done before calling any other Julia C function is to initialize Julia. This is done by calling , which tries to automatically determine Julia's install location. If you need to specify a custom location, or specify which system image to load, use instead.

The second statement in the test program evaluates a Julia statement using a call to .

Before the program terminates, it is strongly recommended to call . The above example program calls this before returning from .

Note

Currently, dynamically linking with the shared library requires passing the option. In Python, this looks like:

Note

If the julia program needs to access symbols from the main executable, it may be necessary to add linker flag at compile time on Linux in addition to the ones generated by described below. This is not necessary when compiling a shared library.

Using julia-config to automatically determine build parameters

The script was created to aid in determining what build parameters are required by a program that uses embedded Julia. This script uses the build parameters and system configuration of the particular Julia distribution it is invoked by to export the necessary compiler flags for an embedding program to interact with that distribution. This script is located in the Julia shared data directory.

Example

On the command line

A simple use of this script is from the command line. Assuming that is located in , it can be invoked on the command line directly and takes any combination of 3 flags:

If the above example source is saved in the file , then the following command will compile it into a running program on Linux and Windows (MSYS2 environment), or if on OS/X, then substitute for .:

Use in Makefiles

But in general, embedding projects will be more complicated than the above, and so the following allows general makefile support as well – assuming GNU make because of the use of the **shell** macro expansions. Additionally, though many times may be found in the directory , this is not necessarily the case, but Julia can be used to locate too, and the makefile can be used to take advantage of that. The above example is extended to use a Makefile:

Now the build command is simply .

33.2 Converting Types

Real applications will not just need to execute expressions, but also return their values to the host program. returns a , which is a pointer to a heap-allocated Julia object. Storing simple data types like in this way is called , and extracting the stored primitive data is called . Our improved sample program that calculates the square root of 2 in Julia and reads back the result in C looks as follows:

In order to check whether is of a specific Julia type, we can use the , , or functions. By typing into the Julia shell we can see that the return type is (in C). To convert the boxed Julia value into a C double the function is used in the above code snippet.

Corresponding functions are used to convert the other way:

As we will see next, boxing is required to call Julia functions with specific arguments.

33.3 Calling Julia Functions

While allows C to obtain the result of a Julia expression, it does not allow passing arguments computed in C to Julia. For this you will need to invoke Julia functions directly, using :

In the first step, a handle to the Julia function is retrieved by calling . The first argument passed to is a pointer to the module in which is defined. Then, the double value is boxed using . Finally, in the last step, the function is called using . , , and functions also exist, to conveniently handle different numbers of arguments. To pass more arguments, use :

Its second argument is an array of arguments and is the number of arguments.

33.4 Memory Management

As we have seen, Julia objects are represented in C as pointers. This raises the question of who is responsible for freeing these objects.

Typically, Julia objects are freed by a garbage collector (GC), but the GC does not automatically know that we are holding a reference to a Julia value from C. This means the GC can free objects out from under you, rendering pointers invalid.

The GC can only run when Julia objects are allocated. Calls like perform allocation, and allocation might also happen at any point in running Julia code. However, it is generally safe to use pointers in between calls. But in order to make sure that values can survive calls, we have to tell Julia that we hold a reference to a Julia value. This can be done using the macros:

The call releases the references established by the previous . Note that is working on the stack, so it must be exactly paired with a before the stack frame is destroyed.

Several Julia values can be pushed at once using the , , and macros. To push an array of Julia values one can use the macro, which can be used as follows:

The garbage collector also operates under the assumption that it is aware of every old-generation object pointing to a young-generation one. Any time a pointer is updated breaking that assumption, it must be signaled to the collector with the (write barrier) function like so:

It is in general impossible to predict which values will be old at runtime, so the write barrier must be inserted after all explicit stores. One notable exception is if the object was just allocated and garbage collection was not run since then. Remember that most functions can sometimes invoke garbage collection.

The write barrier is also necessary for arrays of pointers when updating their data directly. For example:

Manipulating the Garbage Collector

There are some functions to control the GC. In normal use cases, these should not be necessary.

Function	Description
	Force a GC run
	Disable the GC, return previous state as int
	Enable the GC, return previous state as int
	Return current state as int

33.5 Working with Arrays

Julia and C can share array data without copying. The next example will show how this works.

Julia arrays are represented in C by the datatype . Basically, is a struct that contains:

- Information about the datatype
- A pointer to the data block
- Information about the sizes of the array

To keep things simple, we start with a 1D array. Creating an array containing Float64 elements of length 10 is done by:

Alternatively, if you have already allocated the array you can generate a thin wrapper around its data:

The last argument is a boolean indicating whether Julia should take ownership of the data. If this argument is non-zero, the GC will call on the data pointer when the array is no longer referenced.

In order to access the data of x, we can use :

Now we can fill the array:

Now let us call a Julia function that performs an in-place operation on :

By printing the array, one can verify that the elements of are now reversed.

Accessing Returned Arrays

If a Julia function returns an array, the return value of and can be cast to a :

Now the content of can be accessed as before using . As always, be sure to keep a reference to the array while it is in use.

Multidimensional Arrays

Julia's multidimensional arrays are stored in memory in column-major order. Here is some code that creates a 2D array and accesses its properties:

Notice that while Julia arrays use 1-based indexing, the C API uses 0-based indexing (for example in calling) in order to read as idiomatic C code.

33.6 Exceptions

Julia code can throw exceptions. For example, consider:

This call will appear to do nothing. However, it is possible to check whether an exception was thrown:

If you are using the Julia C API from a language that supports exceptions (e.g. Python, C#, C++), it makes sense to wrap each call into with a function that checks whether an exception was thrown, and then rethrows the exception in the host language.

Throwing Julia Exceptions

When writing Julia callable functions, it might be necessary to validate arguments and throw exceptions to indicate errors. A typical type check looks like:

General exceptions can be raised using the functions:

 takes a C string, and is called like :

where in this example is assumed to be an integer.

Chapter 34

Packages

Julia has a built-in package manager for installing add-on functionality written in Julia. It can also install external libraries using your operating system's standard system for doing so, or by compiling from source. The list of registered Julia packages can be found at http://pkg.julialang.org. All package manager commands are found in the module, included in Julia's install.

First we'll go over the mechanics of the family of commands and then we'll provide some guidance on how to get your package registered. Be sure to read the section below on package naming conventions, tagging versions and the importance of a file for when you're ready to add your code to the curated METADATA repository.

34.1 Package Status

The function prints out a summary of the state of packages you have installed. Initially, you'll have no packages installed:

Your package directory is automatically initialized the first time you run a command that expects it to exist – which includes . Here's an example non-trivial set of required and additional packages:

These packages are all on registered versions, managed by . Packages can be in more complicated states, indicated by annotations to the right of the installed package version; we will explain these states and annotations as we encounter them. For programmatic usage, returns a dictionary, mapping installed package names to the version of that package which is installed:

34.2 Adding and Removing Packages

Julia's package manager is a little unusual in that it is declarative rather than imperative. This means that you tell it what you want and it figures out what versions to install (or remove) to satisfy those requirements optimally – and minimally. So rather than installing a package, you just add it to the list of requirements and then "resolve" what needs to be installed. In particular, this means that if some package had been installed because it was needed by a previous version of something you wanted, and a newer version doesn't have that requirement anymore, updating will actually remove that package.

Your package requirements are in the file . You can edit this file by hand and then call to install, upgrade or remove packages to optimally satisfy the requirements, or you can do , which will open in your editor (configured via the or environment variables), and then automatically call afterwards if necessary. If you only want to add or remove the requirement for a single package, you can also use the non-interactive and commands, which add or remove a single requirement to and then call .

You can add a package to the list of requirements with the function, and the package and all the packages that it depends on will be installed:

What this is doing is first adding to your file:

It then runs using these new requirements, which leads to the conclusion that the package should be installed since it is required but not installed. As stated before, you can accomplish the same thing by editing your file by hand and then running yourself:

This is functionally equivalent to calling , except that doesn't change until *after* installation has completed, so if there are problems, will be left as it was before calling . The format of the file is described in Requirements Specification; it allows, among other things, requiring specific ranges of versions of packages.

When you decide that you don't want to have a package around any more, you can use to remove the requirement for it from the file:

Once again, this is equivalent to editing the file to remove the line with each package name on it then running to update the set of installed packages to match. While and are convenient for adding and removing requirements for a single package, when you want to add or remove multiple packages, you can call to manually change the contents of and then update your packages accordingly. does not roll back the contents of if fails – rather, you have to run again to fix the files contents yourself.

Because the package manager uses libgit2 internally to manage the package git repositories, users may run into protocol issues (if behind a firewall, for example), when running . By default, all GitHub-hosted packages wil be accessed via 'https'; this default can be modified by calling . The following command can be run from the command line in order to tell git to use 'https' instead of the 'git' protocol when cloning all repositories, wherever they are hosted:

However, this change will be system-wide and thus the use of is preferable.

> **Note**
>
> The package manager functions also accept the suffix on package names, though the suffix is stripped internally. For example:

34.3 Offline Installation of Packages

For machines with no Internet connection, packages may be installed by copying the package root directory (given by) from a machine with the same operating system and environment.

does the following within the package root directory:

1. Adds the name of the package to .
2. Downloads the package to , then copies the package to the package root directory.
3. Recursively performs step 2 against all the packages listed in the package's file.
4. Runs

Warning

Copying installed packages from a different machine is brittle for packages requiring binary external dependencies. Such packages may break due to differences in operating system versions, build environments, and/or absolute path dependencies.

34.4 Installing Unregistered Packages

Julia packages are simply git repositories, clonable via any of the protocols that git supports, and containing Julia code that follows certain layout conventions. Official Julia packages are registered in the METADATA.jl repository, available at a well-known location [1]. The and commands in the previous section interact with registered packages, but the package manager can install and work with unregistered packages too. To install an unregistered package, use , where is a git URL from which the package can be cloned:

By convention, Julia repository names end with (the additional indicates a "bare" git repository), which keeps them from colliding with repositories for other languages, and also makes Julia packages easy to find in search engines. When packages are installed in your directory, however, the extension is redundant so we leave it off.

If unregistered packages contain a file at the top of their source tree, that file will be used to determine which registered packages the unregistered package depends on, and they will automatically be installed. Unregistered packages participate in the same version resolution logic as registered packages, so installed package versions will be adjusted as necessary to satisfy the requirements of both registered and unregistered packages.

[1] The official set of packages is at https://github.com/JuliaLang/METADATA.jl, but individuals and organizations can easily use a different metadata repository. This allows control which packages are available for automatic installation. One can allow only audited and approved package versions, and make private packages or forks available. See Custom METADATA Repository for details.

34.5 Updating Packages

When package developers publish new registered versions of packages that you're using, you will, of course, want the new shiny versions. To get the latest and greatest versions of all your packages, just do :

The first step of updating packages is to pull new changes to and see if any new registered package versions have been published. After this, attempts to update packages that are checked out on a branch and not dirty (i.e. no changes have been made to files tracked by git) by pulling changes from the package's upstream repository. Upstream changes will only be applied if no merging or rebasing is necessary – i.e. if the branch can be "fast-forwarded". If the branch cannot be fast-forwarded, it is assumed that you're working on it and will update the repository yourself.

Finally, the update process recomputes an optimal set of package versions to have installed to satisfy your top-level requirements and the requirements of "fixed" packages. A package is considered fixed if it is one of the following:

1. **Unregistered:** the package is not in – you installed it with .
2. **Checked out:** the package repo is on a development branch.
3. **Dirty:** changes have been made to files in the repo.

If any of these are the case, the package manager cannot freely change the installed version of the package, so its requirements must be satisfied by whatever other package versions it picks. The combination of top-level requirements in and the requirement of fixed packages are used to determine what should be installed.

You can also update only a subset of the installed packages, by providing arguments to the function. In that case, only the packages provided as arguments and their dependencies will be updated:

This partial update process still computes the new set of package versions according to top-level requirements and "fixed" packages, but it additionally considers all other packages except those explicitly provided, and their dependencies, as fixed.

34.6 Checkout, Pin and Free

You may want to use the version of a package rather than one of its registered versions. There might be fixes or functionality on master that you need that aren't yet published in any registered versions, or you may be a developer of the package and need to make changes on or some other development branch. In such cases, you can do to checkout the branch of or to checkout some other branch:

Immediately after installing with it is on the current most recent registered version – at the time of writing this. Then after running , you can see from the output of that is on an unregistered version greater than , indicated by the ”pseudo-version” number .

When you checkout an unregistered version of a package, the copy of the file in the package repo takes precedence over any requirements registered in , so it is important that developers keep this file accurate and up-to-date, reflecting the actual requirements of the current version of the package. If the file in the package repo is incorrect or missing, dependencies may be removed when the package is checked out. This file is also used to populate newly published versions of the package if you use the API that provides for this (described below).

When you decide that you no longer want to have a package checked out on a branch, you can ”free” it back to the control of the package manager with :

After this, since the package is on a registered version and not on a branch, its version will be updated as new registered versions of the package are published.

If you want to pin a package at a specific version so that calling won't change the version the package is on, you can use the function:

After this, the package will remain pinned at version – or more specifically, at commit , but since versions are permanently associated a given git hash, this is the same thing. works by creating a throw-away branch for the commit you want to pin the package at and then checking that branch out. By default, it pins a package at the current commit, but you can choose a different version by passing a second argument:

Now the package is pinned at commit , which corresponds to version . When you decide to ”unpin” a package and let the package manager update it again, you can use like you would to move off of any branch:

After this, the package is managed by the package manager again, and future calls to will upgrade it to newer versions when they are published. The throw-away branch remains in your local repo, but since git branches are extremely lightweight, this doesn't really matter; if you feel like cleaning them up, you can go into the repo and delete those branches [2].

34.7 Custom METADATA Repository

By default, Julia assumes you will be using the official METADATA.jl repository for downloading and installing packages. You can also provide a different metadata repository location. A common approach is to keep your branch up to date with the Julia official branch and add another branch with your custom packages. You can initialize your local metadata repository using that custom location and branch and then periodically rebase your custom branch with the official branch. In order to use a custom repository and branch, issue the following command:

[2] Packages that aren't on branches will also be marked as dirty if you make changes in the repo, but that's a less common thing to do.

The branch argument is optional and defaults to . Once initialized, a file named in your path will track the branch that your METADATA repository was initialized with. If you want to change branches, you will need to either modify the file directly (be careful!) or remove the directory and re-initialize your METADATA repository using the command.

Chapter 35

Package Development

Julia's package manager is designed so that when you have a package installed, you are already in a position to look at its source code and full development history. You are also able to make changes to packages, commit them using git, and easily contribute fixes and enhancements upstream. Similarly, the system is designed so that if you want to create a new package, the simplest way to do so is within the infrastructure provided by the package manager.

35.1 Initial Setup

Since packages are git repositories, before doing any package development you should setup the following standard global git configuration settings:

where is your actual full name (spaces are allowed between the double quotes) and is your actual email address. Although it isn't necessary to use GitHub to create or publish Julia packages, most Julia packages as of writing this are hosted on GitHub and the package manager knows how to format origin URLs correctly and otherwise work with the service smoothly. We recommend that you create a free account on GitHub and then do:

where is your actual GitHub user name. Once you do this, the package manager knows your GitHub user name and can configure things accordingly. You should also upload your public SSH key to GitHub and set up an SSH agent on your development machine so that you can push changes with minimal hassle. In the future, we will make this system extensible and support other common git hosting options like BitBucket and allow developers to choose their favorite. Since the package development functions has been moved to the PkgDev package, you need to run to access the functions starting with in the document below.

35.2 Making changes to an existing package

Documentation changes

If you want to improve the online documentation of a package, the easiest approach (at least for small changes) is to use GitHub's online editing functionality. First, navigate to the repository's GitHub "home page," find the file (e.g.,) within the repository's folder structure, and click on it. You'll see the contents displayed, along with a small "pencil" icon in the upper right hand corner. Clicking that icon opens the file in edit mode. Make your changes, write a brief summary describing the changes you want to make (this is your *commit message*), and then hit "Propose file change." Your changes will be submitted for consideration by the package owner(s) and collaborators.

For larger documentation changes–and especially ones that you expect to have to update in response to feedback–you might find it easier to use the procedure for code changes described below.

Code changes

Executive summary

Here we assume you've already set up git on your local machine and have a GitHub account (see above). Let's imagine you're fixing a bug in the Images package:

The last line will present you with a link to submit a pull request to incorporate your changes.

Detailed description

If you want to fix a bug or add new functionality, you want to be able to test your changes before you submit them for consideration. You also need to have an easy way to update your proposal in response to the package owner's feedback. Consequently, in this case the strategy is to work locally on your own machine; once you are satisfied with your changes, you submit them for consideration. This process is called a *pull request* because you are asking to "pull" your changes into the project's main repository. Because the online repository can't see the code on your private machine, you first *push* your changes to a publicly-visible location, your own online *fork* of the package (hosted on your own personal GitHub account).

Let's assume you already have the package installed. In the description below, anything starting with or is meant to be typed at the Julia prompt; anything starting with is meant to be typed in julia's shell mode (or using the shell that comes with your operating system). Within Julia, you can combine these two modes:

Now suppose you're ready to make some changes to . While there are several possible approaches, here is one that is widely used:

- From the Julia prompt, type . This ensures you're running the latest code (the branch), rather than just whatever "official release" version you have installed. (If you're planning to fix a bug, at this point it's a good idea to check again whether the bug has already been fixed by someone else. If it has, you can request that a new official release be tagged so that the fix gets distributed to the rest of the community.) If you receive an error , see Dirty packages below.

- Create a branch for your changes: navigate to the package folder (the one that Julia reports from) and (in shell mode) create a new branch using , where might be some descriptive name (e.g.,). By creating a branch, you ensure that you can easily go back and forth between your new work and the current branch (see https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell).

  If you forget to do this step until after you've already made some changes, don't worry: see more detail about branching below.

- Make your changes. Whether it's fixing a bug or adding new functionality, in most cases your change should include updates to both the and folders. If you're fixing a bug, add your minimal example demonstrating the bug (on the current code) to the test suite; by contributing a test for the bug, you ensure that the bug won't accidentally reappear at some later time due to other changes. If you're adding new functionality, creating tests demonstrates to the package owner that you've made sure your code works as intended.

- Run the package's tests and make sure they pass. There are several ways to run the tests:

  - From Julia, run : this will run your tests in a separate (new) process.
  - From Julia, from the package's folder (it's possible the file has a different name, look for one that runs all the tests): this allows you to run the tests repeatedly in the same session without reloading all the package code; for packages that take a while to load, this can be much faster. With this approach, you do have to do some extra work to make changes in the package code.
  - From the shell, run from within the package's folder.

- Commit your changes: see https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository.

- Submit your changes: From the Julia prompt, type . This will push your changes to your GitHub fork, creating it if it doesn't already exist. (If you encounter an error, make sure you've set up your SSH keys.) Julia will then give you a hyperlink; open that link, edit the message, and then click "submit." At that point, the package owner will be notified of your changes and may initiate discussion. (If you are comfortable with git, you can also do these steps manually from the shell.)

- The package owner may suggest additional improvements. To respond to those suggestions, you can easily update the pull request (this only works for changes that have not already been merged; for merged pull requests, make new changes by starting a new branch):

  - If you've changed branches in the meantime, make sure you go back to the same branch with (from shell mode) or (from the Julia prompt).
  - As above, make your changes, run the tests, and commit your changes.
  - From the shell, type . This will add your new commit(s) to the same pull request; you should see them appear automatically on the page holding the discussion of your pull request.

  One potential type of change the owner may request is that you squash your commits. See Squashing below.

Dirty packages

If you can't change branches because the package manager complains that your package is dirty, it means you have some changes that have not been committed. From the shell, use to see what these changes are; you can either discard them () or commit them before switching branches. If you can't easily resolve the problems manually, as a last resort you can delete the entire folder and reinstall a fresh copy with . Naturally, this deletes any changes you've made.

Making a branch *post hoc*

Especially for newcomers to git, one often forgets to create a new branch until after some changes have already been made. If you haven't yet staged or committed your changes, you can create a new branch with just as usual–git will kindly show you that some files have been modified and create the new branch for you. *Your changes have not yet been committed to this new branch*, so the normal work rules still apply.

However, if you've already made a commit to but wish to go back to the official (called), use the following procedure:

- Create a new branch. This branch will hold your changes.

- Make sure everything is committed to this branch.
- . If this fails, *do not* proceed further until you have resolved the problems, or you may lose your changes.
- *Reset* (your current branch) back to an earlier state with (see https://git-scm.com/blog/2011/07/11/reset.html).

This requires a bit more familiarity with git, so it's much better to get in the habit of creating a branch at the outset.

Squashing and rebasing

Depending on the tastes of the package owner (s)he may ask you to "squash" your commits. This is especially likely if your change is quite simple but your commit history looks like this:

This gets into the territory of more advanced git usage, and you're encouraged to do some reading (https://git-scm.com/book/en/v2/Git-Branching-Rebasing). However, a brief summary of the procedure is as follows:

- To protect yourself from error, start from your branch and create a new branch with . Since you started from , this will be a copy. Now go back to the one you intend to modify with .
- From the shell, type .
- To combine commits, change to (for additional options, consult other sources). Save the file and close the editor window.
- Edit the combined commit message.

If the rebase goes badly, you can go back to the beginning to try again like this:

Now let's assume you've rebased successfully. Since your repository has now diverged from the one in your GitHub fork, you're going to have to do a *force push*:

- To make it easy to refer to your GitHub fork, create a "handle" for it with , where the URL comes from the "clone URL" on your GitHub fork's page.
- Force-push to your fork with . The indicates that this should replace the branch found at .

35.3 Creating a new Package

REQUIRE speaks for itself

You should have a file in your package repository, with a bare minimum directive of what Julia version you expect your users to be running for the package to work. Putting a floor on what Julia version your package supports is done by simply adding in this file. While this line is partly informational, it also has the consequence of whether will update code

found in version directories. It will not update code found in version directories beneath the floor of what's specified in your .

As the development version matures, you may find yourself using it more frequently, and wanting your package to support it. Be warned, the development branch of Julia is the land of breakage, and you can expect things to break. When you go about fixing whatever broke your package in the development branch, you will likely find that you just broke your package on the stable version.

There is a mechanism found in the Compat package that will enable you to support both the stable version and breaking changes found in the development version. Should you decide to use this solution, you will need to add to your file. In this case, you will still have in your . The is the floor version of what your package supports.

You might also have no interest in supporting the development version of Julia. Just as you can add a floor to the version you expect your users to be on, you can set an upper bound. In this case, you would put in your file. The at the end of the version number means pre-release versions of that specific version from the very first commit. By setting it as the ceiling, you mean the code supports everything up to but not including the ceiling version.

Another scenario is that you are writing the bulk of the code for your package with Julia and do not want to support the current stable version of Julia. If you choose to do this, simply add to your . Just remember to change the to in your file once is officially released. If you don't edit the dash cruft you are suggesting that you support both the development and stable versions of the same version number! That would be madness. See the Requirements Specification for the full format of .

Lastly, in many cases you may need extra packages for testing. Additional packages which are only required for tests should be specified in the file. This file has the same specification as the standard file.

Guidelines for naming a package

Package names should be sensible to most Julia users, *even to those who are not domain experts*. When you submit your package to METADATA, you can expect a little back and forth about the package name with collaborators, especially if it's ambiguous or can be confused with something other than what it is. During this bike-shedding, it's not uncommon to get a range of *different* name suggestions. These are only suggestions though, with the intent being to keep a tidy namespace in the curated METADATA repository. Since this repository belongs to the entire community, there will likely be a few collaborators who care about your package name. Here are some guidelines to follow in naming your package:

1. Avoid jargon. In particular, avoid acronyms unless there is minimal possibility of confusion.
   - It's ok to say if you're talking about the USA.
   - It's not ok to say , even if you're talking about positive mental attitude.
2. Avoid using in your package name.
   - It is usually clear from context and to your users that the package is a Julia package.
   - Having Julia in the name can imply that the package is connected to, or endorsed by, contributors to the Julia language itself.
3. Packages that provide most of their functionality in association with a new type should have pluralized names.
   - provides the type.
   - provides the type.
   - In contrast, provides no new type, but instead new functionality in the function.
4. Err on the side of clarity, even if clarity seems long-winded to you.

- is a less ambiguous name than or , even though the latter are shorter.

5. A less systematic name may suit a package that implements one of several possible approaches to its domain.
   - Julia does not have a single comprehensive plotting package. Instead, , , and other packages each implement a unique approach based on a particular design philosophy.
   - In contrast, provides a consistent interface to use many well-established sorting algorithms.
6. Packages that wrap external libraries or programs should be named after those libraries or programs.
   - wraps the library, which can be identified easily in a web search.
   - provides an interface to call the MATLAB engine from within Julia.

Generating the package

Suppose you want to create a new Julia package called . To get started, do where is the new package name and is the name of a license that the package generator knows about:

This creates the directory , initializes it as a git repository, generates a bunch of files that all packages should have, and commits them to the repository:

At the moment, the package manager knows about the MIT "Expat" License, indicated by , the Simplified BSD License, indicated by , and version 2.0 of the Apache Software License, indicated by . If you want to use a different license, you can ask us to add it to the package generator, or just pick one of these three and then modify the file after it has been generated.

If you created a GitHub account and configured git to know about it, will set an appropriate origin URL for you. It will also automatically generate a file for using the Travis automated testing service, and an file for using AppVeyor. You will have to enable testing on the Travis and AppVeyor websites for your package repository, but once you've done that, it will already have working tests. Of course, all the default testing does is verify that in Julia works.

Loading Static Non-Julia Files

If your package code needs to load static files which are not Julia code, e.g. an external library or data files, and are located within the package directory, use the macro to determine the directory of the current source file. For example if needs to load , use the following code:

Making Your Package Available

Once you've made some commits and you're happy with how is working, you may want to get some other people to try it out. First you'll need to create the remote repository and push your code to it; we don't yet automatically do this for you, but we will in the future and it's not too hard to figure out [3]. Once you've done this, letting people try out your code is as simple as sending them the URL of the published repo – in this case:

For your package, it will be your GitHub user name and the name of your package, but you get the idea. People you send this URL to can use to install the package and try it out:

Tagging and Publishing Your Package

> **Tip**
>
> If you are hosting your package on GitHub, you can use the attobot integration to handle package registration, tagging and publishing.

Once you've decided that is ready to be registered as an official package, you can add it to your local copy of using :

[3] Installing and using GitHub's "hub" tool is highly recommended. It allows you to do things like run in the package repo and have it automatically created via GitHub's API.

This creates a commit in the repo:

This commit is only locally visible, however. To make it visible to the Julia community, you need to merge your local upstream into the official repo. The command will fork the repository on GitHub, push your changes to your fork, and open a pull request:

Tip

If fails with error:

then you may have encountered an issue from using the GitHub API on multiple systems. The solution is to delete the "Julia Package Manager" personal access token from your Github account and try again.

Other failures may require you to circumvent by creating a pull request on GitHub. See: Publishing METADATA manually below.

Once the package URL for is registered in the official repo, people know where to clone the package from, but there still aren't any registered versions available. You can tag and register it with the command:

This tags in the repo:

It also creates a new version entry in your local repo for :

The command takes an optional second argument that is either an explicit version number object like or one of the symbols , or . These increment the patch, minor or major version number of your package intelligently.

Adding a tagged version of your package will expedite the official registration into METADATA.jl by collaborators. It is strongly recommended that you complete this process, regardless if your package is completely ready for an official release.

As a general rule, packages should be tagged first. Since Julia itself hasn't achieved status, it's best to be conservative in your package's tagged versions.

As with , these changes to aren't available to anyone else until they've been included upstream. Again, use the command, which first makes sure that individual package repos have been tagged, pushes them if they haven't already been, and then opens a pull request to :

Publishing METADATA manually

If fails you can follow these instructions to manually publish your package.

By "forking" the main METADATA repository, you can create a personal copy (of METADATA.jl) under your GitHub account. Once that copy exists, you can push your local changes to your copy (just like any other GitHub project).

1. go to https://github.com/login?return_to=https%3A%2F%2Fgithub.com%2FJuliaLang%2FMETADATA.jl%2Ffork

and create your own fork.

2. add your fork as a remote repository for the METADATA repository on your local computer (in

the terminal where USERNAME is your github username):

1. push your changes to your fork:

2. If all of that works, then go back to the GitHub page for your fork, and click the "pull request"

link.

35.4 Fixing Package Requirements

If you need to fix the registered requirements of an already-published package version, you can do so just by editing the metadata for that version, which will still have the same commit hash – the hash associated with a version is permanent:

Since the commit hash stays the same, the contents of the file that will be checked out in the repo will **not** match the requirements in after such a change; this is unavoidable. When you fix the requirements in for a previous version of a package, however, you should also fix the file in the current version of the package.

35.5 Requirements Specification

The file, the file inside packages, and the package files use a simple line-based format to express the ranges of package versions which need to be installed. Package and files should also include the range of versions of the package is expected to work with. Additionally, packages can include a file to specify additional packages which are only required for testing.

Here's how these files are parsed and interpreted.

- Everything after a mark is stripped from each line as a comment.
- If nothing but whitespace is left, the line is ignored.
- If there are non-whitespace characters remaining, the line is a requirement and the is split on whitespace into words.

The simplest possible requirement is just the name of a package name on a line by itself:

This requirement is satisfied by any version of the package. The package name can be followed by zero or more version numbers in ascending order, indicating acceptable intervals of versions of that package. One version opens an interval, while the next closes it, and the next opens a new interval, and so on; if an odd number of version numbers are given, then arbitrarily large versions will satisfy; if an even number of version numbers are given, the last one is an upper limit on acceptable version numbers. For example, the line:

```

```

is satisfied by any version of greater than or equal to . Suffixing a version with allows any pre-release versions as well. For example:

```

```

is satisfied by pre-release versions such as or , or by any version greater than or equal to .

This requirement entry:

```

```

is satisfied by versions from up to, but not including . If you want to indicate that any version will do, you will want to write:

```

```

If you want to start accepting versions after , you can write:

```

```

If a requirement line has leading words that begin with , it is a system-dependent requirement. If your system matches these system conditionals, the requirement is included, if not, the requirement is ignored. For example:

```

```

will require the package only on systems where the operating system is OS X. The system conditions that are currently supported are (hierarchically):

-
  -
  -
    - *
-

The condition is satisfied on all UNIX systems, including Linux and BSD. Negated system conditionals are also supported by adding a after the leading . Examples:

```

```

The first condition applies to any system but Windows and the second condition applies to any UNIX system besides OS X.

Runtime checks for the current version of Julia can be made using the built-in variable, which is of type . Such code is occasionally necessary to keep track of new or deprecated functionality between various releases of Julia. Examples of runtime checks:

```

```

See the section on version number literals for a more complete description.

Chapter 36

Profiling

The module provides tools to help developers improve the performance of their code. When used, it takes measurements on running code, and produces output that helps you understand how much time is spent on individual line(s). The most common usage is to identify "bottlenecks" as targets for optimization.

 implements what is known as a "sampling" or statistical profiler. It works by periodically taking a backtrace during the execution of any task. Each backtrace captures the currently-running function and line number, plus the complete chain of function calls that led to this line, and hence is a "snapshot" of the current state of execution.

If much of your run time is spent executing a particular line of code, this line will show up frequently in the set of all backtraces. In other words, the "cost" of a given line–or really, the cost of the sequence of function calls up to and including this line–is proportional to how often it appears in the set of all backtraces.

A sampling profiler does not provide complete line-by-line coverage, because the backtraces occur at intervals (by default, 1 ms on Unix systems and 10 ms on Windows, although the actual scheduling is subject to operating system load). Moreover, as discussed further below, because samples are collected at a sparse subset of all execution points, the data collected by a sampling profiler is subject to statistical noise.

Despite these limitations, sampling profilers have substantial strengths:

- You do not have to make any modifications to your code to take timing measurements (in contrast to the alternative instrumenting profiler).
- It can profile into Julia's core code and even (optionally) into C and Fortran libraries.
- By running "infrequently" there is very little performance overhead; while profiling, your code can run at nearly native speed.

For these reasons, it's recommended that you try using the built-in sampling profiler before considering any alternatives.

36.1 Basic usage

Let's work with a simple test case:

It's a good idea to first run the code you intend to profile at least once (unless you want to profile Julia's JIT-compiler):

Now we're ready to profile this function:

To see the profiling results, there is a graphical browser available, but here we'll use the text-based display that comes with the standard library:

Each line of this display represents a particular spot (line number) in the code. Indentation is used to indicate the nested sequence of function calls, with more-indented lines being deeper in the sequence of calls. In each line, the first "field" is the number of backtraces (samples) taken *at this line or in any functions executed by this line.* The second field is the file name and line number and the third field is the function name. Note that the specific line numbers may change as Julia's code changes; if you want to follow along, it's best to run this example yourself.

In this example, we can see that the top level function called is in the file . This is the function that runs the REPL when you launch Julia. If you examine line 97 of , you'll see this is where the function is called. This is the function that evaluates what you type at the REPL, and since we're working interactively these functions were invoked when we entered . The next line reflects actions taken in the macro.

The first line shows that 80 backtraces were taken at line 73 of , but it's not that this line was "expensive" on its own: the third line reveals that all 80 of these backtraces were actually triggered inside its call to , and so on. To find out which operations are actually taking the time, we need to look deeper in the call chain.

The first "important" line in this output is this one:

refers to the fact that we defined in the REPL, rather than putting it in a file; if we had used a file, this would show the file name. The shows that the function was the first expression evaluated in this REPL session. Line 2 of contains the call to , and there were 52 (out of 80) backtraces that occurred at this line. Below that, you can see a call to inside .

A little further down, you see:

Line 3 of contains the call to , and there were 28 (out of 80) backtraces taken here. Below that, you can see the specific places in that carry out the time-consuming operations in the function for this type of input data.

Overall, we can tentatively conclude that generating the random numbers is approximately twice as expensive as finding the maximum element. We could increase our confidence in this result by collecting more samples:

In general, if you have samples collected at a line, you can expect an uncertainty on the order of (barring other sources of noise, like how busy the computer is with other tasks). The major exception to this rule is garbage collection, which runs infrequently but tends to be quite expensive. (Since Julia's garbage collector is written in C, such events can be detected using the output mode described below, or by using ProfileView.jl.)

This illustrates the default "tree" dump; an alternative is the "flat" dump, which accumulates counts independent of their nesting:

If your code has recursion, one potentially-confusing point is that a line in a "child" function can accumulate more counts than there are total backtraces. Consider the following function definitions:

If you were to profile , and a backtrace was taken while it was executing , the backtrace would look like this:

Consequently, this child function gets 3 counts, even though the parent only gets one. The ”tree” representation makes this much clearer, and for this reason (among others) is probably the most useful way to view the results.

36.2 Accumulation and clearing

Results from accumulate in a buffer; if you run multiple pieces of code under , then will show you the combined results. This can be very useful, but sometimes you want to start fresh; you can do so with .

36.3 Options for controlling the display of profile results

has more options than we've described so far. Let's see the full declaration:

Let's first discuss the two positional arguments, and later the keyword arguments:

- – Allows you to save the results to a buffer, e.g. a file, but the default is to print to (the console).
- – Contains the data you want to analyze; by default that is obtained from , which pulls out the backtraces from a pre-allocated buffer. For example, if you want to profile the profiler, you could say:

The keyword arguments can be any combination of:

- – Introduced above, determines whether backtraces are printed with (default,) or without () indentation indicating tree structure.
- – If , backtraces from C and Fortran code are shown (normally they are excluded). Try running the introductory example with . This can be extremely helpful in deciding whether it's Julia code or C code that is causing a bottleneck; setting also improves the interpretability of the nesting, at the cost of longer profile dumps.
- – Some lines of code contain multiple operations; for example, contains both an array reference () and a sum operation. These correspond to different lines in the generated machine code, and hence there may be two or more different addresses captured during backtraces on this line. lumps them together, and is probably what you typically want, but you can generate an output separately for each unique instruction pointer with .
- – Limits frames at a depth higher than in the format.

- – Controls the order in format. (default) sorts by the source line, whereas sorts in order of number of collected samples.
- – Limits frames that are below the heuristic noise floor of the sample (only applies to format). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which , where is the number of samples on this line, and is the number of samples for the callee.
- – Limits frames with less than occurrences.

File/function names are sometimes truncated (with), and indentation is truncated with a at the beginning, where is the number of extra spaces that would have been inserted, had there been room. If you want a complete profile of deeply-nested code, often a good idea is to save to a file using a wide in an :

36.4 Configuration

just accumulates backtraces, and the analysis happens when you call . For a long-running computation, it's entirely possible that the pre-allocated buffer for storing backtraces will be filled. If that happens, the backtraces stop but your computation continues. As a consequence, you may miss some important profiling data (you will get a warning when that happens).

You can obtain and configure the relevant parameters this way:

is the total number of instruction pointers you can store, with a default value of . If your typical backtrace is 20 instruction pointers, then you can collect 50000 backtraces, which suggests a statistical uncertainty of less than 1%. This may be good enough for most applications.

Consequently, you are more likely to need to modify , expressed in seconds, which sets the amount of time that Julia gets between snapshots to perform the requested computations. A very long-running job might not need frequent backtraces. The default setting is . Of course, you can decrease the delay as well as increase it; however, the overhead of profiling grows once the delay becomes similar to the amount of time needed to take a backtrace (~30 microseconds on the author's laptop).

Chapter 37

Memory allocation analysis

One of the most common techniques to improve performance is to reduce memory allocation. The total amount of allocation can be measured with and , and specific lines triggering allocation can often be inferred from profiling via the cost of garbage collection that these lines incur. However, sometimes it is more efficient to directly measure the amount of memory allocated by each line of code.

To measure allocation line-by-line, start Julia with the command-line option, for which you can choose (the default, do not measure allocation), (measure memory allocation everywhere except Julia's core code), or (measure memory allocation at each line of Julia code). Allocation gets measured for each line of compiled code. When you quit Julia, the cumulative results are written to text files with appended after the file name, residing in the same directory as the source file. Each line lists the total number of bytes allocated. The package contains some elementary analysis tools, for example to sort the lines in order of number of bytes allocated.

In interpreting the results, there are a few important details. Under the setting, the first line of any function directly called from the REPL will exhibit allocation due to events that happen in the REPL code itself. More significantly, JIT-compilation also adds to allocation counts, because much of Julia's compiler is written in Julia (and compilation usually requires memory allocation). The recommended procedure is to force compilation by executing all the commands you want to analyze, then call to reset all allocation counters. Finally, execute the desired commands and quit Julia to trigger the generation of the files.

Chapter 38

Stack Traces

The module provides simple stack traces that are both human readable and easy to use programmatically.

38.1 Viewing a stack trace

The primary function used to obtain a stack trace is :

Calling returns a vector of s. For ease of use, the alias can be used in place of . (Examples with indicate that output may vary depending on how the code is run.)

Note that when calling you'll typically see a frame with . When calling from the REPL you'll also have a few extra frames in the stack from , usually looking something like this:

38.2 Extracting useful information

Each contains the function name, file name, line number, lambda info, a flag indicating whether the frame has been inlined, a flag indicating whether it is a C function (by default C functions do not appear in the stack trace), and an integer representation of the pointer returned by :

This makes stack trace information available programmatically for logging, error handling, and more.

38.3 Error handling

While having easy access to information about the current state of the callstack can be helpful in many places, the most obvious application is in error handling and debugging.

You may notice that in the example above the first stack frame points points at line 4, where is called, rather than line 2, where *bad_function* is called, and 's frame is missing entirely. This is understandable, given that is called from the context of the *catch*. While in this example it's fairly easy to find the actual source of the error, in complex cases tracking down the source of the error becomes nontrivial.

This can be remedied by calling instead of . Instead of returning callstack information for the current context, returns stack information for the context of the most recent exception:

Notice that the stack trace now indicates the appropriate line number and the missing frame.

38.4 Comparison with

A call to returns a vector of , which may then be passed into for translation:

Notice that the vector returned by had 21 pointers, while the vector returned by only has 5. This is because, by default, removes any lower-level C functions from the stack. If you want to include stack frames from C calls, you can do it like this:

Individual pointers returned by can be translated into s by passing them into :

Chapter 39

Performance Tips

In the following sections, we briefly go through a few techniques that can help make your Julia code run as fast as possible.

39.1 Avoid global variables

A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables. Variables should be local, or passed as arguments to functions, whenever possible.

Any code that is performance critical or being benchmarked should be inside a function.

We find that global names are frequently constants, and declaring them as such greatly improves performance:

Uses of non-constant globals can be optimized by annotating their types at the point of use:

Writing functions is better style. It leads to more reusable code and clarifies what steps are being done, and what their inputs and outputs are.

> **Note**
>
> All code in the REPL is evaluated in global scope, so a variable defined and assigned at toplevel will be a **global** variable.

In the following REPL session:

is equivalent to:

so all the performance issues discussed previously apply.

39.2 Measure performance with and pay attention to memory allocation

A useful tool for measuring performance is the macro. The following example illustrates good working style:

On the first call (), gets compiled. (If you've not yet used in this session, it will also compile functions needed for timing.) You should not take the results of this run seriously. For the second run, note that in addition to reporting the time, it also indicated that a large amount of memory was allocated. This is the single biggest advantage of vs. functions like and , which only report time.

Unexpected memory allocation is almost always a sign of some problem with your code, usually a problem with type-stability. Consequently, in addition to the allocation itself, it's very likely that the code generated for your function is far from optimal. Take such indications seriously and follow the advice below.

For more serious benchmarking, consider the BenchmarkTools.jl package which evaluates the function multiple times in order to reduce noise.

As a teaser, an improved version of this function allocates no memory (the allocation reported below is due to running the macro in global scope) and has an order of magnitude faster execution after the first call:

Below you'll learn how to spot the problem with and how to fix it.

In some situations, your function may need to allocate memory as part of its operation, and this can complicate the simple picture above. In such cases, consider using one of the tools below to diagnose problems, or write a version of your function that separates allocation from its algorithmic aspects (see Pre-allocating outputs).

39.3 Tools

Julia and its package ecosystem includes tools that may help you diagnose problems and improve the performance of your code:

- Profiling allows you to measure the performance of your running code and identify lines that serve as bottlenecks. For complex projects, the ProfileView package can help you visualize your profiling results.
- Unexpectedly-large memory allocations–as reported by , , or the profiler (through calls to the garbage-collection routines)–hint that there might be issues with your code. If you don't see another reason for the allocations, suspect a type problem. You can also start Julia with the option and examine the resulting files to see information about where those allocations occur. See Memory allocation analysis.
- generates a representation of your code that can be helpful in finding expressions that result in type uncertainty. See below.
- The Lint package can also warn you of certain types of programming errors.

39.4 Avoid containers with abstract type parameters

When working with parameterized types, including arrays, it is best to avoid parameterizing with abstract types where possible.

Consider the following:

Because is a an array of abstract type , it must be able to hold any value. Since objects can be of arbitrary size and structure, must be represented as an array of pointers to individually allocated objects. Because will always be a , we should instead, use:

which will create a contiguous block of 64-bit floating-point values that can be manipulated efficiently.

See also the discussion under Parametric Types.

39.5 Type declarations

In many languages with optional type declarations, adding declarations is the principal way to make code run faster. This is *not* the case in Julia. In Julia, the compiler generally knows the types of all function arguments, local variables, and expressions. However, there are a few specific instances where declarations are helpful.

Avoid fields with abstract type

Types can be declared without specifying the types of their fields:

This allows to be of any type. This can often be useful, but it does have a downside: for objects of type , the compiler will not be able to generate high-performance code. The reason is that the compiler uses the types of objects, not their values, to determine how to build code. Unfortunately, very little can be inferred about an object of type :

and have the same type, yet their underlying representation of data in memory is very different. Even if you stored just numeric values in field , the fact that the memory representation of a differs from a also means that the CPU needs to handle them using two different kinds of instructions. Since the required information is not available in the type, such decisions have to be made at run-time. This slows performance.

You can do better by declaring the type of . Here, we are focused on the case where might be any one of several types, in which case the natural solution is to use parameters. For example:

This is a better choice than

because the first version specifies the type of from the type of the wrapper object. For example:

The type of field can be readily determined from the type of , but not from the type of . Indeed, in it's possible to change the type of field :

In contrast, once is constructed, the type of cannot change:

The fact that the type of is known from 's type–coupled with the fact that its type cannot change mid-function–allows the compiler to generate highly-optimized code for objects like but not for objects like .

Of course, all of this is true only if we construct with a concrete type. We can break this by explicitly constructing it with an abstract type:

For all practical purposes, such objects behave identically to those of .

It's quite instructive to compare the sheer amount code generated for a simple function

using

For reasons of length the results are not shown here, but you may wish to try this yourself. Because the type is fully-specified in the first case, the compiler doesn't need to generate any code to resolve the type at run-time. This results in shorter and faster code.

Avoid fields with abstract containers

The same best practices also work for container types:

For example:

For , the object is fully-specified by its type and parameters, so the compiler can generate optimized functions. In most instances, this will probably suffice.

While the compiler can now do its job perfectly well, there are cases where *you* might wish that your code could do different things depending on the *element type* of . Usually the best way to achieve this is to wrap your specific operation (here,) in a separate function:

This keeps things simple, while allowing the compiler to generate optimized code in all cases.

However, there are cases where you may need to declare different versions of the outer function for different element types of . You could do it like this:

This works fine for , but we'd also have to write explicit versions for or other abstract types. To prevent such tedium, you can use two parameters in the declaration of :

Note the somewhat surprising fact that doesn't appear in the declaration of field , a point that we'll return to in a moment. With this approach, one can write functions such as:

Note

Because we can only define for , and any unspecified parameters are arbitrary, the first function above could have been written more succinctly as

As you can see, with this approach it's possible to specialize on both the element type and the array type .

However, there's one remaining hole: we haven't enforced that has element type , so it's perfectly possible to construct an object like this:

To prevent this, we can add an inner constructor:

The inner constructor requires that the element type of be .

Annotate values taken from untyped locations

It is often convenient to work with data structures that may contain values of any type (arrays of type). But, if you're using one of these structures and happen to know the type of an element, it helps to share this knowledge with the compiler:

Here, we happened to know that the first element of would be an . Making an annotation like this has the added benefit that it will raise a run-time error if the value is not of the expected type, potentially catching certain bugs earlier.

In the case that the type of is not known precisely, can be declared via . The use of the function allows to be any object convertible to an (such as), thus increasing the genericity of the code by loosening the type requirement. Notice that itself needs a type annotation in this context in order to achieve type stability. This is because the compiler cannot deduce the type of the return value of a function, even , unless the types of all the function's arguments are known.

Declare types of keyword arguments

Keyword arguments can have declared types:

Functions are specialized on the types of keyword arguments, so these declarations will not affect performance of code inside the function. However, they will reduce the overhead of calls to the function that include keyword arguments.

Functions with keyword arguments have near-zero overhead for call sites that pass only positional arguments.

Passing dynamic lists of keyword arguments, as in , can be slow and should be avoided in performance-sensitive code.

39.6 Break functions into multiple definitions

Writing a function as many small definitions allows the compiler to directly call the most applicable code, or even inline it.

Here is an example of a ”compound function” that should really be written as multiple definitions:

This can be written more concisely and efficiently as:

39.7 Write ”type-stable” functions

When possible, it helps to ensure that a function always returns a value of the same type. Consider the following definition:

Although this seems innocent enough, the problem is that is an integer (of type) and might be of any type. Thus, depending on the value of , this function might return a value of either of two types. This behavior is allowed, and may be desirable in some cases. But it can easily be fixed as follows:

There is also a function, and a more general function, which returns converted to the type of .

39.8 Avoid changing the type of a variable

An analogous ”type-stability” problem exists for variables used repeatedly within a function:

Local variable starts as an integer, and after one loop iteration becomes a floating-point number (the result of operator). This makes it more difficult for the compiler to optimize the body of the loop. There are several possible fixes:

- Initialize with
- Declare the type of :
- Use an explicit conversion:
- Initialize with the first loop iteration, to , then loop

39.9 Separate kernel functions (aka, function barriers)

Many functions follow a pattern of performing some set-up work, and then running many iterations to perform a core computation. Where possible, it is a good idea to put these core computations in separate functions. For example, the following contrived function returns an array of a randomly-chosen type:

This should be written as:

Julia's compiler specializes code for argument types at function boundaries, so in the original implementation it does not know the type of during the loop (since it is chosen randomly). Therefore the second version is generally faster since the inner loop can be recompiled as part of for different types of .

The second form is also often better style and can lead to more code reuse.

This pattern is used in several places in the standard library. For example, see in , or the function, which we could have used instead of writing our own .

Functions like occur when dealing with data of uncertain type, for example data loaded from an input file that might contain either integers, floats, strings, or something else.

39.10 Types with values-as-parameters

Let's say you want to create an -dimensional array that has size 3 along each axis. Such arrays can be created like this:

This approach works very well: the compiler can figure out that is an because it knows the type of the fill value () and the dimensionality (). This implies that the compiler can generate very efficient code for any future usage of in the same function.

But now let's say you want to write a function that creates a 3×3×... array in arbitrary dimensions; you might be tempted to write a function

This works, but (as you can verify for yourself using) the problem is that the output type cannot be inferred: the argument is a *value* of type , and type-inference does not (and cannot) predict its value in advance. This means that code using the output of this function has to be conservative, checking the type on each access of ; such code will be very slow.

Now, one very good way to solve such problems is by using the function-barrier technique. However, in some cases you might want to eliminate the type-instability altogether. In such cases, one approach is to pass the dimensionality as a parameter, for example through (see "Value types"):

Julia has a specialized version of that accepts a instance as the second parameter; by passing as a type-parameter, you make its ”value” known to the compiler. Consequently, this version of allows the compiler to predict the return type.

However, making use of such techniques can be surprisingly subtle. For example, it would be of no help if you called from a function like this:

Here, you’ve created the same problem all over again: the compiler can’t guess what is, so it doesn’t know the *type* of . Attempting to use , but doing so incorrectly, can easily make performance *worse* in many situations. (Only in situations where you’re effectively combining with the function-barrier trick, to make the kernel function more efficient, should code like the above be used.)

An example of correct usage of would be:

In this example, is passed as a parameter, so its ”value” is known to the compiler. Essentially, works only when is either hard-coded/literal () or already specified in the type-domain.

39.11 The dangers of abusing multiple dispatch (aka, more on types with values-as-parameters)

Once one learns to appreciate multiple dispatch, there’s an understandable tendency to go crazy and try to use it for everything. For example, you might imagine using it to store information, e.g.

and then dispatch on objects like .

This might be worthwhile when the following are true:

- You require CPU-intensive processing on each , and it becomes vastly more efficient if you know the and at compile time.
- You have homogenous lists of the same type of to process, so that you can store them all in an .

When the latter holds, a function processing such a homogenous array can be productively specialized: Julia knows the type of each element in advance (all objects in the container have the same concrete type), so Julia can ”look up” the correct method calls when the function is being compiled (obviating the need to check at run-time) and thereby emit efficient code for processing the whole list.

When these do not hold, then it’s likely that you’ll get no benefit; worse, the resulting ”combinatorial explosion of types” will be counterproductive. If has a different type than , Julia has to look up the type at run-time, search for the appropriate method in method tables, decide (via type intersection) which one matches, determine whether it has been JIT-compiled yet (and do so if not), and then make the call. In essence, you’re asking the full type- system and JIT-compilation machinery to basically execute the equivalent of a switch statement or dictionary lookup in your own code.

Some run-time benchmarks comparing (1) type dispatch, (2) dictionary lookup, and (3) a "switch" statement can be found on the mailing list.

Perhaps even worse than the run-time impact is the compile-time impact: Julia will compile specialized functions for each different ; if you have hundreds or thousands of such types, then every function that accepts such an object as a parameter (from a custom function you might write yourself, to the generic function in the standard library) will have hundreds or thousands of variants compiled for it. Each of these increases the size of the cache of compiled code, the length of internal lists of methods, etc. Excess enthusiasm for values-as-parameters can easily waste enormous resources.

39.12 Access arrays in memory order, along columns

Multidimensional arrays in Julia are stored in column-major order. This means that arrays are stacked one column at a time. This can be verified using the function or the syntax as shown below (notice that the array is ordered , not):

This convention for ordering arrays is common in many languages like Fortran, Matlab, and R (to name a few). The alternative to column-major ordering is row-major ordering, which is the convention adopted by C and Python () among other languages. Remembering the ordering of arrays can have significant performance effects when looping over arrays. A rule of thumb to keep in mind is that with column-major arrays, the first index changes most rapidly. Essentially this means that looping will be faster if the inner-most loop index is the first to appear in a slice expression.

Consider the following contrived example. Imagine we wanted to write a function that accepts a and returns a square with either the rows or the columns filled with copies of the input vector. Assume that it is not important whether rows or columns are filled with these copies (perhaps the rest of the code can be easily adapted accordingly). We could conceivably do this in at least four ways (in addition to the recommended call to the built-in):

Now we will time each of these functions using the same random by input vector:

Notice that is much faster than . This is expected because respects the column-based memory layout of the and fills it one column at a time. Additionally, is much faster than because it follows our rule of thumb that the first element to appear in a slice expression should be coupled with the inner-most loop.

39.13 Pre-allocating outputs

If your function returns an or some other complex type, it may have to allocate memory. Unfortunately, oftentimes allocation and its converse, garbage collection, are substantial bottlenecks.

Sometimes you can circumvent the need to allocate memory on each function call by preallocating the output. As a trivial example, compare

with

Timing results:

Preallocation has other advantages, for example by allowing the caller to control the ”output” type from an algorithm. In the example above, we could have passed a rather than an , had we so desired.

Taken to its extreme, pre-allocation can make your code uglier, so performance measurements and some judgment may be required. However, for ”vectorized” (element-wise) functions, the convenient syntax can be used for in-place operations with fused loops and no temporary arrays (see the dot syntax for vectorizing functions).

39.14 More dots: Fuse vectorized operations

Julia has a special dot syntax that converts any scalar function into a ”vectorized” function call, and any operator into a ”vectorized” operator, with the special property that nested ”dot calls” are *fusing*: they are combined at the syntax level into a single loop, without allocating temporary arrays. If you use and similar assignment operators, the result can also be stored in-place in a pre-allocated array (see above).

In a linear-algebra context, this means that even though operations like and are defined, it can be advantageous to instead use and because the resulting loops can be fused with surrounding computations. For example, consider the two functions:

Both and compute the same thing. However, (defined with the help of the macro) is significantly faster when applied to an array:

That is, is three times faster and allocates 1/7 the memory of , because each and operation in allocates a new temporary array and executes in a separate loop. (Of course, if you just do then it is as fast as in this example, but in many contexts it is more convenient to just sprinkle some dots in your expressions rather than defining a separate function for each vectorized operation.)

39.15 Consider using views for slices

In Julia, an array "slice" expression like creates a copy of that data (except on the left-hand side of an assignment, where assigns in-place to that portion of). If you are doing many operations on the slice, this can be good for performance because it is more efficient to work with a smaller contiguous copy than it would be to index into the original array. On the other hand, if you are just doing a few simple operations on the slice, the cost of the allocation and copy operations can be substantial.

An alternative is to create a "view" of the array, which is an array object (a) that actually references the data of the original array in-place, without making a copy. (If you write to a view, it modifies the original array's data as well.) This can be done for individual slices by calling , or more simply for a whole expression or block of code by putting in front of that expression. For example:

Notice both the 3× speedup and the decreased memory allocation of the version of the function.

39.16 Copying data is not always bad

Arrays are stored contiguously in memory, lending themselves to CPU vectorization and fewer memory accesses due to caching. These are the same reasons that it is recommended to access arrays in column-major order (see above). Irregular access patterns and non-contiguous views can drastically slow down computations on arrays because of non-sequential memory access.

Copying irregularly-accessed data into a contiguous array before operating on it can result in a large speedup, such as in the example below. Here, a matrix and a vector are being accessed at 800,000 of their randomly-shuffled indices before being multiplied. Copying the views into plain arrays speeds the multiplication by more than a factor of 2 even with the cost of the copying operation.

Provided there is enough memory for the copies, the cost of copying the view to an array is far outweighed by the speed boost from doing the matrix multiplication on a contiguous array.

39.17 Avoid string interpolation for I/O

When writing data to a file (or other I/O device), forming extra intermediate strings is a source of overhead. Instead of:

use:

The first version of the code forms a string, then writes it to the file, while the second version writes values directly to the file. Also notice that in some cases string interpolation can be harder to read. Consider:

versus:

39.18 Optimize network I/O during parallel execution

When executing a remote function in parallel:

is faster than:

The former results in a single network round-trip to every worker, while the latter results in two network calls - first by the and the second due to the (or even a). The / is also being executed serially resulting in an overall poorer performance.

39.19 Fix deprecation warnings

A deprecated function internally performs a lookup in order to print a relevant warning only once. This extra lookup can cause a significant slowdown, so all uses of deprecated functions should be modified as suggested by the warnings.

39.20 Tweaks

These are some minor points that might help in tight inner loops.

- Avoid unnecessary arrays. For example, instead of use .
- Use instead of for complex . In general, try to rewrite code to use instead of for complex arguments.
- Use for truncating division of integers instead of , instead of , and instead of .

39.21 Performance Annotations

Sometimes you can enable better optimization by promising certain program properties.

- Use to eliminate array bounds checking within expressions. Be certain before doing this. If the subscripts are ever out of bounds, you may suffer crashes or silent corruption.
- Use to allow floating point optimizations that are correct for real numbers, but lead to differences for IEEE numbers. Be careful when doing this, as this may change numerical results. This corresponds to the option of clang.
- Write in front of loops that are amenable to vectorization. **This feature is experimental** and could change or disappear in future versions of Julia.

Note: While needs to be placed directly in front of a loop, both and can be applied to several statements at once, e.g. using ... , or even to a whole function.

Here is an example with both and markup:

On a computer with a 2.4GHz Intel Core i5 processor, this produces:

(measures the performance, and larger numbers are better.) The range for a loop should be a one-dimensional range. A variable used for accumulating, such as in the example, is called a *reduction variable.* By using , you are asserting several properties of the loop:

- It is safe to execute iterations in arbitrary or overlapping order, with special consideration for reduction variables.
- Floating-point operations on reduction variables can be reordered, possibly causing different results than without .
- No iteration ever waits on another iteration to make forward progress.

A loop containing , , or will cause a compile-time error.

Using merely gives the compiler license to vectorize. Whether it actually does so depends on the compiler. To actually benefit from the current implementation, your loop should have the following additional properties:

- The loop must be an innermost loop.
- The loop body must be straight-line code. This is why is currently needed for all array accesses. The compiler can sometimes turn short , , and expressions into straight-line code, if it is safe to evaluate all operands unconditionally. Consider using instead of in the loop if it is safe to do so.
- Accesses must have a stride pattern and cannot be ”gathers” (random-index reads) or ”scatters” (random-index writes).
- The stride should be unit stride.
- In some simple cases, for example with 2-3 arrays accessed in a loop, the LLVM auto-vectorization may kick in automatically, leading to no further speedup with .

Here is an example with all three kinds of markup. This program first calculates the finite difference of a one-dimensional array, and then evaluates the L2-norm of the result:

On a computer with a 2.7 GHz Intel Core i7 processor, this produces:

Here, the option disables the macro, so that we can compare results.

In this case, the speedup due to is a factor of about 3.7. This is unusually large – in general, the speedup will be smaller. (In this particular example, the working set of the benchmark is small enough to fit into the L1 cache of the processor, so that memory access latency does not play a role, and computing time is dominated by CPU usage. In many real world programs this is not the case.) Also, in this case this optimization does not change the result – in general, the result will be slightly different. In some cases, especially for numerically unstable algorithms, the result can be very different.

The annotation re-arranges floating point expressions, e.g. changing the order of evaluation, or assuming that certain special cases (inf, nan) cannot occur. In this case (and on this particular computer), the main difference is that the expression in the function is hoisted out of the loop (i.e. calculated outside the loop), as if one had written . In the loop, the expression then becomes , which is much faster to evaluate. Of course, both the actual optimization that is applied by the compiler as well as the resulting speedup depend very much on the hardware. You can examine the change in generated code by using Julia's function.

39.22 Treat Subnormal Numbers as Zeros

Subnormal numbers, formerly called denormal numbers, are useful in many contexts, but incur a performance penalty on some hardware. A call grants permission for floating-point operations to treat subnormal inputs or outputs as zeros, which may improve performance on some hardware. A call enforces strict IEEE behavior for subnormal numbers.

Below is an example where subnormals noticeably impact performance on some hardware:

This example generates many subnormal numbers because the values in become an exponentially decreasing curve, which slowly flattens out over time.

Treating subnormals as zeros should be used with caution, because doing so breaks some identities, such as implies :

In some applications, an alternative to zeroing subnormal numbers is to inject a tiny bit of noise. For example, instead of initializing with zeros, initialize it with:

39.23

The macro (or its function variant) can sometimes be helpful in diagnosing type-related problems. Here's an example:

Interpreting the output of , like that of its cousins , , , and , takes a little practice. Your code is being presented in form that has been partially digested on its way to generating compiled machine code. Most of the expressions are annotated by a type, indicated by the (where might be , for example). The most important characteristic of is that non-concrete types are displayed in red; in the above example, such output is shown in all-caps.

The top part of the output summarizes the type information for the different variables internal to the function. You can see that , one of the variables you created, is a , due to the type-instability of . There is another variable, , which you can see also has the same type.

The next lines represent the body of . The lines starting with a number followed by a colon (,) are labels, and represent targets for jumps (via) in your code. Looking at the body, you can see that has been *inlined* into –everything before comes from code defined in .

Starting at , the variable is defined, and again annotated as a type. Next, we see that the compiler created the temporary variable to hold the result of . Because a times *either* an or yields a , all type-instability ends here. The net result is that will not be type-unstable in its output, even if some of the intermediate computations are type-unstable.

How you use this information is up to you. Obviously, it would be far and away best to fix to be type-stable: if you did so, all of the variables in would be concrete, and its performance would be optimal. However, there are circumstances where this kind of *ephemeral* type instability might not matter too much: for example, if is never used in isolation, the fact that 's output is type-stable (for inputs) will shield later code from the propagating effects of type instability. This is particularly relevant in cases where fixing the type instability is difficult or impossible: for example, currently it's not possible to infer the return type of an anonymous function. In such cases, the tips above (e.g., adding type annotations and/or breaking up functions) are your best tools to contain the "damage" from type instability.

The following examples may help you interpret expressions marked as containing non-leaf types:

- Function body ending in

  - Interpretation: function with unstable return type
  - Suggestion: make the return value type-stable, even if you have to annotate it

-
  - Interpretation: call to a type-unstable function
  - Suggestion: fix the function, or if necessary annotate the return value

-
  - Interpretation: accessing elements of poorly-typed arrays
  - Suggestion: use arrays with better-defined types, or if necessary annotate the type of individual element accesses

-
  - Interpretation: getting a field that is of non-leaf type. In this case, had a field . But needs the dimension , too, to be a concrete type.
  - Suggestion: use concrete types like or , where is now a parameter of

Chapter 40

Workflow Tips

Here are some tips for working with Julia efficiently.

40.1 REPL-based workflow

As already elaborated in Interacting With Julia, Julia's REPL provides rich functionality that facilitates an efficient interactive workflow. Here are some tips that might further enhance your experience at the command line.

A basic editor/REPL workflow

The most basic Julia workflows involve using a text editor in conjunction with the command line. A common pattern includes the following elements:

- **Put code under development in a temporary module.** Create a file, say , and include within it

- **Put your test code in another file.** Create another file, say , which begins with

  and includes tests for the contents of . The value of using versus is that you can call instead of having to restart the REPL when your definitions change. Of course, the cost is the need to prepend to uses of names defined in your module. (You can lower that cost by keeping your module name short.)

  Alternatively, you can wrap the contents of your test file in a module, as

  The advantage is that you can now do in your test code and can therefore avoid prepending everywhere. The disadvantage is that code can no longer be selectively copied to the REPL without some tweaking.

- **Lather. Rinse. Repeat.** Explore ideas at the command prompt. Save good ideas in . Occasionally restart the REPL, issuing

Simplify initialization

To simplify restarting the REPL, put project-specific initialization code in a file, say , which you can run on startup by issuing the command:

If you further add the following to your file

then calling from that directory will run the initialization code without the additional command line argument.

40.2 Browser-based workflow

It is also possible to interact with a Julia REPL in the browser via IJulia. See the package home for details.

Chapter 41

Style Guide

The following sections explain a few aspects of idiomatic Julia coding style. None of these rules are absolute; they are only suggestions to help familiarize you with the language and to help you choose among alternative designs.

41.1 Write functions, not just scripts

Writing code as a series of steps at the top level is a quick way to get started solving a problem, but you should try to divide a program into functions as soon as possible. Functions are more reusable and testable, and clarify what steps are being done and what their inputs and outputs are. Furthermore, code inside functions tends to run much faster than top level code, due to how Julia's compiler works.

It is also worth emphasizing that functions should take arguments, instead of operating directly on global variables (aside from constants like).

41.2 Avoid writing overly-specific types

Code should be as generic as possible. Instead of writing:

it's better to use available generic functions:

The second version will convert to an appropriate type, instead of always the same type.

This style point is especially relevant to function arguments. For example, don't declare an argument to be of type or if it really could be any integer, expressed with the abstract type . In fact, in many cases you can omit the argument type altogether, unless it is needed to disambiguate from other method definitions, since a will be thrown anyway if a type is passed that does not support any of the requisite operations. (This is known as duck typing.)

For example, consider the following definitions of a function that returns one plus its argument:

The last definition of handles any type supporting (which returns 1 in the same type as , which avoids unwanted type promotion) and the function with those arguments. The key thing to realize is that there is *no performance penalty* to defining *only* the general , because Julia will automatically compile specialized versions as needed. For example, the first time you call , Julia will automatically compile a specialized function for arguments, with the call to replaced by its inlined value . Therefore, the first three definitions of above are completely redundant with the fourth definition.

41.3 Handle excess argument diversity in the caller

Instead of:

use:

This is better style because does not really accept numbers of all types; it really needs s.

One issue here is that if a function inherently requires integers, it might be better to force the caller to decide how non-integers should be converted (e.g. floor or ceiling). Another issue is that declaring more specific types leaves more ”space” for future method definitions.

41.4 Append to names of functions that modify their arguments

Instead of:

use:

The Julia standard library uses this convention throughout and contains examples of functions with both copying and modifying forms (e.g., and), and others which are just modifying (e.g., , ,). It is typical for such functions to also return the modified array for convenience.

41.5 Avoid strange type s

Types such as are often a sign that some design could be cleaner.

41.6 Avoid type Unions in fields

When creating a type such as:

ask whether the option for to be (of type) is really necessary. Here are some alternatives to consider:

- Find a safe default value to initialize with
- Introduce another type that lacks
- If there are many fields like , store them in a dictionary
- Determine whether there is a simple rule for when is . For example, often the field will start as but get initialized at some well-defined point. In that case, consider leaving it undefined at first.
- If really needs to hold no value at some times, define it as instead, as this guarantees type-stability in the code accessing this field (see Nullable types).

41.7 Avoid elaborate container types

It is usually not much help to construct arrays like the following:

In this case is better. It is also more helpful to the compiler to annotate specific uses (e.g.) than to try to pack many alternatives into one type.

41.8 Use naming conventions consistent with Julia's

- modules and type names use capitalization and camel case: , .
- functions are lowercase (,) and, when readable, with multiple words squashed together (,). When necessary, use underscores as word separators. Underscores are also used to indicate a combination of concepts (as a more efficient implementation of) or as modifiers ().
- conciseness is valued, but avoid abbreviation (rather than) as it becomes difficult to remember whether and how particular words are abbreviated.

If a function name requires multiple words, consider whether it might represent more than one concept and might be better split into pieces.

41.9 Don't overuse try-catch

It is better to avoid errors than to rely on catching them.

41.10 Don't parenthesize conditions

Julia doesn't require parens around conditions in and . Write:

instead of:

41.11 Don't overuse

Splicing function arguments can be addictive. Instead of , use simply , which already concatenates arrays. is better than , but since is already iterable it is often even better to leave it alone, and not convert it to an array.

41.12 Don't use unnecessary static parameters

A function signature:

should be written as:

instead, especially if is not used in the function body. Even if is used, it can be replaced with if convenient. There is no performance difference. Note that this is not a general caution against static parameters, just against uses where they are not needed.

Note also that container types, specifically may need type parameters in function calls. See the FAQ Avoid fields with abstract containers for more information.

41.13 Avoid confusion about whether something is an instance or a type

Sets of definitions like the following are confusing:

Decide whether the concept in question will be written as or , and stick to it.

The preferred style is to use instances by default, and only add methods involving later if they become necessary to solve some problem.

If a type is effectively an enumeration, it should be defined as a single (ideally immutable struct or primitive) type, with the enumeration values being instances of it. Constructors and conversions can check whether values are valid. This design is preferred over making the enumeration an abstract type, with the "values" as subtypes.

41.14 Don't overuse macros

Be aware of when a macro could really be a function instead.

Calling inside a macro is a particularly dangerous warning sign; it means the macro will only work when called at the top level. If such a macro is written as a function instead, it will naturally have access to the run-time values it needs.

41.15 Don't expose unsafe operations at the interface level

If you have a type that uses a native pointer:

don't write definitions like the following:

The problem is that users of this type can write without realizing that the operation is unsafe, and then be susceptible to memory bugs.

Such a function should either check the operation to ensure it is safe, or have somewhere in its name to alert callers.

41.16 Don't overload methods of base container types

It is possible to write definitions like the following:

This would provide custom showing of vectors with a specific new element type. While tempting, this should be avoided. The trouble is that users will expect a well-known type like to behave in a certain way, and overly customizing its behavior can make it harder to work with.

41.17 Avoid type piracy

"Type piracy" refers to the practice of extending or redefining methods in Base or other packages on types that you have not defined. In some cases, you can get away with type piracy with little ill effect. In extreme cases, however, you can even crash Julia (e.g. if your method extension or redefinition causes invalid input to be passed to a). Type piracy can complicate reasoning about code, and may introduce incompatibilities that are hard to predict and diagnose.

As an example, suppose you wanted to define multiplication on symbols in a module:

The problem is that now any other module that uses will also see this definition. Since is defined in Base and is used by other modules, this can change the behavior of unrelated code unexpectedly. There are several alternatives here, including using a different function name, or wrapping the s in another type that you define.

Sometimes, coupled packages may engage in type piracy to separate features from definitions, especially when the packages were designed by collaborating authors, and when the definitions are reusable. For example, one package might provide some types useful for working with colors; another package could define methods for those types that enable conversions between color spaces. Another example might be a package that acts as a thin wrapper for some C code, which another package might then pirate to implement a higher-level, Julia-friendly API.

41.18 Be careful with type equality

You generally want to use and for testing types, not . Checking types for exact equality typically only makes sense when comparing to a known concrete type (e.g.), or if you *really, really* know what you're doing.

41.19 Do not write

Since higher-order functions are often called with anonymous functions, it is easy to conclude that this is desirable or even necessary. But any function can be passed directly, without being "wrapped" in an anonymous function. Instead of writing , write .

41.20 Avoid using floats for numeric literals in generic code when possible

If you write generic code which handles numbers, and which can be expected to run with many different numeric type arguments, try using literals of a numeric type that will affect the arguments as little as possible through promotion.

For example,

while

As you can see, the second version, where we used an literal, preserved the type of the input argument, while the first didn't. This is because e.g. , and promotion happens with the multiplication. Similarly, literals are less type disruptive than literals, but more disruptive than s:

Thus, use literals when possible, with for literal non-integer numbers, in order to make it easier to use your code.

Chapter 42

Frequently Asked Questions

42.1 Sessions and the REPL

How do I delete an object in memory?

Julia does not have an analog of MATLAB's function; once a name is defined in a Julia session (technically, in module), it is always present.

If memory usage is your concern, you can always replace objects with ones that consume less memory. For example, if is a gigabyte-sized array that you no longer need, you can free the memory with . The memory will be released the next time the garbage collector runs; you can force this to happen with .

How can I modify the declaration of a type in my session?

Perhaps you've defined a type and then realize you need to add a new field. If you try this at the REPL, you get the error:

```

```

Types in module cannot be redefined.

While this can be inconvenient when you are developing new code, there's an excellent workaround. Modules can be replaced by redefining them, and so if you wrap all your new code inside a module you can redefine types and constants. You can't import the type names into and then expect to be able to redefine them there, but you can use the module name to resolve the scope. In other words, while developing you might use a workflow something like this:

```

```

42.2 Functions

I passed an argument to a function, modified it inside that function, but on the outside, the variable is still unchanged. Why?

Suppose you call a function like this:

In Julia, the binding of a variable cannot be changed by passing as an argument to a function. When calling in the above example, is a newly created variable, bound initially to the value of , i.e. ; then is rebound to the constant , while the variable of the outer scope is left untouched.

But here is a thing you should pay attention to: suppose is bound to an object of type (or any other *mutable* type). From within the function, you cannot ”unbind” from this Array, but you can change its content. For example:

Here we created a function , that assigns to the first element of the passed array (bound to at the call site, and bound to within the function). Notice that, after the function call, is still bound to the same array, but the content of that array changed: the variables and were distinct bindings refering to the same mutable object.

Can I use or inside a function?

No, you are not allowed to have a or statement inside a function. If you want to import a module but only use its symbols inside a specific function or set of functions, you have two options:

1. Use :

This loads the module and defines a variable that refers to the module, but does not import any of the other symbols from the module into the current namespace. You refer to the symbols by their qualified names etc.

2. Wrap your function in a module:

This imports all the symbols from , but only inside the module .

What does the operator do?

The two uses of the operator: slurping and splatting

Many newcomers to Julia find the use of operator confusing. Part of what makes the operator confusing is that it means two different things depending on context.

combines many arguments into one argument in function definitions

In the context of function definitions, the operator is used to combine many different arguments into a single argument. This use of for combining many different arguments into a single argument is called slurping:

If Julia were a language that made more liberal use of ASCII characters, the slurping operator might have been written as instead of .

splits one argument into many different arguments in function calls

In contrast to the use of the operator to denote slurping many different arguments into one argument when defining a function, the operator is also used to cause a single function argument to be split apart into many different arguments when used in the context of a function call. This use of is called splatting:

If Julia were a language that made more liberal use of ASCII characters, the splatting operator might have been written as instead of .

42.3 Types, type declarations, and constructors

What does ”type-stable” mean?

It means that the type of the output is predictable from the types of the inputs. In particular, it means that the type of the output cannot vary depending on the *values* of the inputs. The following code is *not* type-stable:

It returns either an or a depending on the value of its argument. Since Julia can't predict the return type of this function at compile-time, any computation that uses it will have to guard against both types possibly occurring, making generation of fast machine code difficult.

Why does Julia give a for certain seemingly-sensible operations?

Certain operations make mathematical sense but result in errors:

This behavior is an inconvenient consequence of the requirement for type-stability. In the case of , most users want to give a real number, and would be unhappy if it produced the complex number . One could write the function to switch to a complex-valued output only when passed a negative number (which is what does in some other languages), but then the result would not be type-stable and the function would have poor performance.

In these and other cases, you can get the result you want by choosing an *input type* that conveys your willingness to accept an *output type* in which the result can be represented:

Why does Julia use native machine integer arithmetic?

Julia uses machine arithmetic for integer computations. This means that the range of values is bounded and wraps around at either end so that adding, subtracting and multiplying integers can overflow or underflow, leading to some results that can be unsettling at first:

Clearly, this is far from the way mathematical integers behave, and you might think it less than ideal for a high-level programming language to expose this to the user. For numerical work where efficiency and transparency are at a premium, however, the alternatives are worse.

One alternative to consider would be to check each integer operation for overflow and promote results to bigger integer types such as or in the case of overflow. Unfortunately, this introduces major overhead on every integer operation (think incrementing a loop counter) – it requires emitting code to perform run-time overflow checks after arithmetic instructions and branches to handle potential overflows. Worse still, this would cause every computation involving integers to be type-unstable. As we mentioned above, type-stability is crucial for effective generation of efficient code. If you can't count on the results of integer operations being integers, it's impossible to generate fast, simple code the way C and Fortran compilers do.

A variation on this approach, which avoids the appearance of type instability is to merge the and types into a single hybrid integer type, that internally changes representation when a result no longer fits into the size of a machine integer. While this superficially avoids type-instability at the level of Julia code, it just sweeps the problem under the rug by foisting all of the same difficulties onto the C code implementing this hybrid integer type. This approach *can* be made to work and can even be made quite fast in many cases, but has several drawbacks. One problem is that the in-memory representation of integers and arrays of integers no longer match the natural representation used by C, Fortran and other languages with native machine integers. Thus, to interoperate with those languages, we would ultimately need to introduce native integer types anyway. Any unbounded representation of integers cannot have a fixed number of bits, and thus cannot be stored inline in an array with fixed-size slots – large integer values will always require separate heap-allocated storage. And of course, no matter how clever a hybrid integer implementation one uses, there are always performance traps – situations where performance degrades unexpectedly. Complex representation, lack of interoperability with C and Fortran, the inability to represent integer arrays without additional heap storage, and unpredictable performance characteristics make even the cleverest hybrid integer implementations a poor choice for high-performance numerical work.

An alternative to using hybrid integers or promoting to BigInts is to use saturating integer arithmetic, where adding to the largest integer value leaves it unchanged and likewise for subtracting from the smallest integer value. This is precisely what Matlab™ does:

At first blush, this seems reasonable enough since 9223372036854775807 is much closer to 9223372036854775808 than -9223372036854775808 is and integers are still represented with a fixed size in a natural way that is compatible with C and Fortran. Saturated integer arithmetic, however, is deeply problematic. The first and most obvious issue is that this is not the way machine integer arithmetic works, so implementing saturated operations requires emitting instructions after each machine integer operation to check for underflow or overflow and replace the result with or as appropriate. This alone expands each integer operation from a single, fast instruction into half a dozen instructions, probably including branches. Ouch. But it gets worse – saturating integer arithmetic isn't associative. Consider this Matlab computation:

This makes it hard to write many basic integer algorithms since a lot of common techniques depend on the fact that machine addition with overflow *is* associative. Consider finding the midpoint between integer values and in Julia using the expression :

See? No problem. That's the correct midpoint between 2^62 and 2^63, despite the fact that is -4611686018427387904. Now try it in Matlab:

Oops. Adding a operator to Matlab wouldn't help, because saturation that occurs when adding and has already destroyed the information necessary to compute the correct midpoint.

Not only is lack of associativity unfortunate for programmers who cannot rely it for techniques like this, but it also defeats almost anything compilers might want to do to optimize integer arithmetic. For example, since Julia integers use normal machine integer arithmetic, LLVM is free to aggressively optimize simple little functions like . The machine code for this function is just this:

The actual body of the function is a single instruction, which computes the integer multiply and add at once. This is even more beneficial when gets inlined into another function:

Since the call to gets inlined, the loop body ends up being just a single instruction. Next, consider what happens if we make the number of loop iterations fixed:

Because the compiler knows that integer addition and multiplication are associative and that multiplication distributes over addition – neither of which is true of saturating arithmetic – it can optimize the entire loop down to just a multiply and an add. Saturated arithmetic completely defeats this kind of optimization since associativity and distributivity can fail at each loop iteration, causing different outcomes depending on which iteration the failure occurs in. The compiler can unroll the loop, but it cannot algebraically reduce multiple operations into fewer equivalent operations.

The most reasonable alternative to having integer arithmetic silently overflow is to do checked arithmetic everywhere, raising errors when adds, subtracts, and multiplies overflow, producing values that are not value-correct. In this blog post, Dan Luu analyzes this and finds that rather than the trivial cost that this approach should in theory have, it ends

up having a substantial cost due to compilers (LLVM and GCC) not gracefully optimizing around the added overflow checks. If this improves in the future, we could consider defaulting to checked integer arithmetic in Julia, but for now, we have to live with the possibility of overflow.

What are the possible causes of an during remote execution?

As the error states, an immediate cause of an on a remote node is that a binding by that name does not exist. Let us explore some of the possible causes.

The closure carries a reference to , and since is unavailable on node 2, an is thrown.

Globals under modules other than are not serialized by value to the remote node. Only a reference is sent. Functions which create global bindings (except under) may cause an to be thrown later.

In the above example, defined on all nodes. However the call to created a new global binding on the local node, but this was not found on node 2 resulting in an error.

Note that this does not apply to globals created under module . Globals under module are serialized and new bindings created under on the remote node.

This does not apply to or declarations. However, anonymous functions bound to global variables are serialized as can be seen below.

42.4 Packages and Modules

What is the difference between "using" and "importall"?

There is only one difference, and on the surface (syntax-wise) it may seem very minor. The difference between and is that with you need to say to extend module Foo's function bar with a new method, but with or , you only need to say and it automatically extends module Foo's function bar.

If you use , then and become equivalent. If you use , then they are different.

The reason this is important enough to have been given separate syntax is that you don't want to accidentally extend a function that you didn't know existed, because that could easily cause a bug. This is most likely to happen with a method that takes a common type like a string or integer, because both you and the other module could define a method to handle such a common type. If you use , then you'll replace the other module's implementation of with your new implementation, which could easily do something completely different (and break all/many future usages of the other functions in module Foo that depend on calling bar).

42.5 Nothingness and missing values

How does "null" or "nothingness" work in Julia?

Unlike many languages (for example, C and Java), Julia does not have a "null" value. When a reference (variable, object field, or array element) is uninitialized, accessing it will immediately throw an error. This situation can be detected using the function.

Some functions are used only for their side effects, and do not need to return a value. In these cases, the convention is to return the value , which is just a singleton object of type . This is an ordinary type with no fields; there is nothing special about it except for this convention, and that the REPL does not print anything for it. Some language constructs that would not otherwise have a value also yield , for example .

For situations where a value exists only sometimes (for example, missing statistical data), it is best to use the type, which allows specifying the type of a missing value.

The empty tuple () is another form of nothingness. But, it should not really be thought of as nothing but rather a tuple of zero values.

In code written for Julia prior to version 0.4 you may occasionally see , which is quite different. It is the empty (or "bottom") type, a type with no values and no subtypes (except itself). This is now written as (an empty union type). You will generally not need to use this type.

42.6 Memory

Why does allocate memory when and are arrays?

In Julia, gets replaced during parsing by . For arrays, this has the consequence that, rather than storing the result in the same location in memory as , it allocates a new array to store the result.

While this behavior might surprise some, the choice is deliberate. The main reason is the presence of immutable objects within Julia, which cannot change their value once created. Indeed, a number is an immutable object; the statements do not modify the meaning of , they modify the value bound to . For an immutable, the only way to change the value is to reassign it.

To amplify a bit further, consider the following function:

After a call like , you would get the expected result: . However, now suppose that , when used with matrices, instead mutated the left hand side. There would be two problems:

- For general square matrices, cannot be implemented without temporary storage: gets computed and stored on the left hand side before you're done using it on the right hand side.
- Suppose you were willing to allocate a temporary for the computation (which would eliminate most of the point of making work in-place); if you took advantage of the mutability of , then this function would behave differently for mutable vs. immutable inputs. In particular, for immutable , after the call you'd have (in general) , but for mutable you'd have .

Because supporting generic programming is deemed more important than potential performance optimizations that can be achieved by other means (e.g., using explicit loops), operators like and work by rebinding new values.

42.7 Asynchronous IO and concurrent synchronous writes

Why do concurrent writes to the same stream result in inter-mixed output?

While the streaming I/O API is synchronous, the underlying implementation is fully asynchronous.

Consider the printed output from the following:

This is happening because, while the call is synchronous, the writing of each argument yields to other tasks while waiting for that part of the I/O to complete.

and "lock" the stream during a call. Consequently changing to in the above example results in:

You can lock your writes with a like this:

42.8 Julia Releases

Do I want to use a release, beta, or nightly version of Julia?

You may prefer the release version of Julia if you are looking for a stable code base. Releases generally occur every 6 months, giving you a stable platform for writing code.

You may prefer the beta version of Julia if you don't mind being slightly behind the latest bugfixes and changes, but find the slightly faster rate of changes more appealing. Additionally, these binaries are tested before they are published to ensure they are fully functional.

You may prefer the nightly version of Julia if you want to take advantage of the latest updates to the language, and don't mind if the version available today occasionally doesn't actually work.

Finally, you may also consider building Julia from source for yourself. This option is mainly for those individuals who are comfortable at the command line, or interested in learning. If this describes you, you may also be interested in reading our guidelines for contributing.

Links to each of these download types can be found on the download page at https://julialang.org/downloads/. Note that not all versions of Julia are available for all platforms.

When are deprecated functions removed?

Deprecated functions are removed after the subsequent release. For example, functions marked as deprecated in the 0.1 release will not be available starting with the 0.2 release.

Chapter 43

Noteworthy Differences from other Languages

43.1 Noteworthy differences from MATLAB

Although MATLAB users may find Julia's syntax familiar, Julia is not a MATLAB clone. There are major syntactic and functional differences. The following are some noteworthy differences that may trip up Julia users accustomed to MATLAB:

- Julia arrays are indexed with square brackets, .
- Julia arrays are assigned by reference. After , changing elements of will modify as well.
- Julia values are passed and assigned by reference. If a function modifies an array, the changes will be visible in the caller.
- Julia does not automatically grow arrays in an assignment statement. Whereas in MATLAB can create the array and can grow it into , the corresponding Julia statement throws an error if the length of is less than 5 or if this statement is the first use of the identifier . Julia has and , which grow s much more efficiently than MATLAB's .
- The imaginary unit is represented in Julia as , not or as in MATLAB.
- In Julia, literal numbers without a decimal point (such as) create integers instead of floating point numbers. Arbitrarily large integer literals are supported. As a result, some operations such as will throw a domain error as the result is not an integer (see the FAQ entry on domain errors for details).
- In Julia, multiple values are returned and assigned as tuples, e.g. or . MATLAB's , which is often used in MATLAB to do optional work based on the number of returned values, does not exist in Julia. Instead, users can use optional and keyword arguments to achieve similar capabilities.
- Julia has true one-dimensional arrays. Column vectors are of size , not . For example, makes a 1-dimensional array.
- In Julia, will always construct a 3-element array containing , and .
  - To concatenate in the first ("vertical") dimension use either or separate with semicolons ().
  - To concatenate in the second ("horizontal") dimension use either or separate with spaces ().
  - To construct block matrices (concatenating in the first two dimensions), use either or combine spaces and semicolons ().
- In Julia, and construct objects. To construct a full vector like in MATLAB, use . Generally, there is no need to call though. will act like a normal array in most cases but is more efficient because it lazily computes its values. This pattern of creating specialized objects instead of full arrays is used frequently, and is also seen in functions such as , or with iterators such as , and . The special objects can mostly be used as if they were normal arrays.

- Functions in Julia return values from their last expression or the keyword instead of listing the names of variables to return in the function definition (see The return Keyword for details).
- A Julia script may contain any number of functions, and all definitions will be externally visible when the file is loaded. Function definitions can be loaded from files outside the current working directory.
- In Julia, reductions such as , , and are performed over every element of an array when called with a single argument, as in , even if has more than one dimension.
- In Julia, functions such as that operate column-wise by default (is equivalent to) do not have special behavior for arrays; the argument is returned unmodified since it still performs . To sort a matrix like a vector, use .
- In Julia, parentheses must be used to call a function with zero arguments, like in and .
- Julia discourages the used of semicolons to end statements. The results of statements are not automatically printed (except at the interactive prompt), and lines of code do not need to end with semicolons. or can be used to print specific output.
- In Julia, if and are arrays, logical comparison operations like do not return an array of booleans. Instead, use , and similarly for the other boolean operators like , and .
- In Julia, the operators , , and () perform the bitwise operations equivalent to , , and respectively in MATLAB, and have precedence similar to Python's bitwise operators (unlike C). They can operate on scalars or element-wise across arrays and can be used to combine logical arrays, but note the difference in order of operations: parentheses may be required (e.g., to select elements of equal to 1 or 2 use).
- In Julia, the elements of a collection can be passed as arguments to a function using the splat operator , as in .
- Julia's returns singular values as a vector instead of as a dense diagonal matrix.
- In Julia, is not used to continue lines of code. Instead, incomplete expressions automatically continue onto the next line.
- In both Julia and MATLAB, the variable is set to the value of the last expression issued in an interactive session. In Julia, unlike MATLAB, is not set when Julia code is run in non-interactive mode.
- Julia's s do not support dynamically adding fields at runtime, unlike MATLAB's es. Instead, use a .
- In Julia each module has its own global scope/namespace, whereas in MATLAB there is just one global scope.
- In MATLAB, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression or in the statement to modify in-place. In contrast, Julia provides the higher order functions and , allowing users to write and as alternatives to the corresponding transliterations and . Using reduces the use of temporary arrays.
- The analogue of extracting (or "dereferencing") all elements of a cell array, e.g. in in MATLAB, is written using the splat operator in Julia, e.g. as .

43.2 Noteworthy differences from R

One of Julia's goals is to provide an effective language for data analysis and statistical programming. For users coming to Julia from R, these are some noteworthy differences:

- Julia's single quotes enclose characters, not strings.
- Julia can create substrings by indexing into strings. In R, strings must be converted into character vectors before creating substrings.

- In Julia, like Python but unlike R, strings can be created with triple quotes . This syntax is convenient for constructing strings that contain line breaks.
- In Julia, varargs are specified using the splat operator , which always follows the name of a specific variable, unlike R, for which can occur in isolation.
- In Julia, modulus is , not . in Julia is the remainder operator.
- In Julia, not all data structures support logical indexing. Furthermore, logical indexing in Julia is supported only with vectors of length equal to the object being indexed. For example:
  - In R, is equivalent to .
  - In R, is equivalent to .
  - In Julia, throws a .
  - In Julia, produces .
- Like many languages, Julia does not always allow operations on vectors of different lengths, unlike R where the vectors only need to share a common index range. For example, is valid R but the equivalent will throw an error in Julia.
- Julia's takes the function first, then its arguments, unlike in R. Similarly Julia's equivalent of in R is where the function is the first argument.
- Multivariate apply in R, e.g. , can be written as in Julia. Equivalently Julia offers a shorter dot syntax for vectorizing functions .
- Julia uses to denote the end of conditional blocks, like , loop blocks, like / , and functions. In lieu of the one-line , Julia allows statements of the form , and . Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. .
- In Julia, , and are not assignment operators.
- Julia's creates an anonymous function, like Python.
- Julia constructs vectors using brackets. Julia's is the equivalent of R's .
- Julia's operator can perform matrix multiplication, unlike in R. If and are matrices, then denotes a matrix multiplication in Julia, equivalent to R's . In R, this same notation would perform an element-wise (Hadamard) product. To get the element-wise multiplication operation, you need to write in Julia.
- Julia performs matrix transposition using the operator and conjugated transposition using the operator. Julia's is therefore equivalent to R's .
- Julia does not require parentheses when writing statements or / loops: use instead of and instead of .
- Julia does not treat the numbers and as Booleans. You cannot write in Julia, because statements accept only booleans. Instead, you can write , , or .
- Julia does not provide and . Instead, use for and for .
- Julia is careful to distinguish scalars, vectors and matrices. In R, and are the same. In Julia, they can not be used interchangeably. One potentially confusing result of this is that for vectors and is a 1-element vector, not a scalar. To get a scalar, use .
- Julia's and are not like R's.

- Julia cannot assign to the results of function calls on the left hand side of an assignment operation: you cannot write .
- Julia discourages populating the main namespace with functions. Most statistical functionality for Julia is found in packages under the JuliaStats organization. For example:
  - Functions pertaining to probability distributions are provided by the Distributions package.
  - The DataFrames package provides data frames.
  - Generalized linear models are provided by the GLM package.
- Julia provides tuples and real hash tables, but not R-style lists. When returning multiple items, you should typically use a tuple: instead of , use .
- Julia encourages users to write their own types, which are easier to use than S3 or S4 objects in R. Julia's multiple dispatch system means that and act like R's and .
- In Julia, values are passed and assigned by reference. If a function modifies an array, the changes will be visible in the caller. This is very different from R and allows new functions to operate on large data structures much more efficiently.
- In Julia, vectors and matrices are concatenated using , and , not , and like in R.
- In Julia, a range like is not shorthand for a vector like in R, but is a specialized that is used for iteration without high memory overhead. To convert a range into a vector, use .
- Julia's and are the equivalent of and respectively in R, but both arguments need to have the same dimensions. While and replace and in R, there are important differences.
- Julia's , , , and are different from their counterparts in R. They all accept one or two arguments. The first argument is an iterable collection such as an array. If there is a second argument, then this argument indicates the dimensions, over which the operation is carried out. For instance, let in Julia and be the same matrix in R. Then gives the same result as , but is a row vector containing the sum over each column and is a column vector containing the sum over each row. This contrasts to the behavior of R, where and . If the second argument is a vector, then it specifies all the dimensions over which the sum is performed, e.g., . It should be noted that there is no error checking regarding the second argument.
- Julia has several functions that can mutate their arguments. For example, it has both and .
- In R, performance requires vectorization. In Julia, almost the opposite is true: the best performing code is often achieved by using devectorized loops.
- Julia is eagerly evaluated and does not support R-style lazy evaluation. For most users, this means that there are very few unquoted expressions or column names.
- Julia does not support the type.
- Julia lacks the equivalent of R's or .
- In Julia, does not require parentheses.
- In R, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression or in the statement to modify in-place. In contrast, Julia provides the higher order functions and , allowing users to write and as alternatives to the corresponding transliterations and . Using reduces the use of temporary arrays.

43.3 Noteworthy differences from Python

- Julia requires to end a block. Unlike Python, Julia has no keyword.
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
- Julia's slice indexing includes the last element, unlike in Python. in Julia is in Python.
- Julia does not support negative indexes. In particular, the last element of a list or array is indexed with in Julia, not as in Python.
- Julia's , , , etc. blocks are terminated by the keyword. Indentation level is not significant as it is in Python.
- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
- Julia arrays are column major (Fortran ordered) whereas NumPy arrays are row major (C-ordered) by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to NumPy (see relevant section of Performance Tips).
- Julia's updating operators (e.g. , , ...) are *not in-place* whereas NumPy's are. This means doesn't change values in , it rather rebinds the name to the result of the right-hand side , which is a new array. For in-place operation, use (see also dot operators), explicit loops, or .
- Julia evaluates default values of function arguments every time the method is invoked, unlike in Python where the default values are evaluated only once when the function is defined. For example, the function returns a new random number every time it is invoked without argument. On the other hand, the function returns every time it is called as .
- In Julia is the remainder operator, whereas in Python it is the modulus.

43.4 Noteworthy differences from C/C++

- Julia arrays are indexed with square brackets, and can have more than one dimension . This syntax is not just syntactic sugar for a reference to a pointer or address as in C/C++. See the Julia documentation for the syntax for array construction (it has changed between versions).
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
- Julia arrays are assigned by reference. After , changing elements of will modify as well. Updating operators like do not operate in-place, they are equivalent to which rebinds the left-hand side to the result of the right-hand side expression.
- Julia arrays are column major (Fortran ordered) whereas C/C++ arrays are row major ordered by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to C/C++ (see relevant section of Performance Tips).
- Julia values are passed and assigned by reference. If a function modifies an array, the changes will be visible in the caller.
- In Julia, whitespace is significant, unlike C/C++, so care must be taken when adding/removing whitespace from a Julia program.

- In Julia, literal numbers without a decimal point (such as) create signed integers, of type , but literals too large to fit in the machine word size will automatically be promoted to a larger size type, such as (if is), , or the arbitrarily large type. There are no numeric literal suffixes, such as , , , , to indicate unsigned and/or signed vs. unsigned. Decimal literals are always signed, and hexadecimal literals (which start with like C/C++), are unsigned. Hexadecimal literals also, unlike C/C++/Java and unlike decimal literals in Julia, have a type based on the *length* of the literal, including leading 0s. For example, and have type , and have type , then literals with 5 to 8 hex digits have type , 9 to 16 hex digits type and 17 to 32 hex digits type . This needs to be taken into account when defining hexadecimal masks, for example is very different from . 64 bit and 32 bit bit literals are expressed as and respectively. Floating point literals are rounded (and not promoted to the type) if they can not be exactly represented. Floating point literals are closer in behavior to C/C++. Octal (prefixed with) and binary (prefixed with) literals are also treated as unsigned.
- String literals can be delimited with either or , delimited literals can contain characters without quoting it like String literals can have values of other variables or expressions interpolated into them, indicated by or , which evaluates the variable name or the expression in the context of the function.
- indicates a number, and not a single-line comment (which is in Julia)
- indicates the start of a multiline comment, and ends it.
- Functions in Julia return values from their last expression(s) or the keyword. Multiple values can be returned from functions and assigned as tuples, e.g. or , instead of having to pass pointers to values as one would have to do in C/C++ (i.e. .
- Julia does not require the use of semicolons to end statements. The results of expressions are not automatically printed (except at the interactive prompt, i.e. the REPL), and lines of code do not need to end with semicolons. or can be used to print specific output. In the REPL, can be used to suppress output. also has a different meaning within , something to watch out for. can be used to separate expressions on a single line, but are not strictly necessary in many cases, and are more an aid to readability.
- In Julia, the operator () performs the bitwise XOR operation, i.e. in C/C++. Also, the bitwise operators do not have the same precedence as C/++, so parenthesis may be required.
- Julia's is exponentiation (pow), not bitwise XOR as in C/C++ (use , or , in Julia)
- Julia has two right-shift operators, and . performs an arithmetic shift, always performs a logical shift, unlike C/C++, where the meaning of depends on the type of the value being shifted.
- Julia's creates an anonymous function, it does not access a member via a pointer.
- Julia does not require parentheses when writing statements or / loops: use instead of and instead of .
- Julia does not treat the numbers and as Booleans. You cannot write in Julia, because statements accept only booleans. Instead, you can write , , or .
- Julia uses to denote the end of conditional blocks, like , loop blocks, like / , and functions. In lieu of the one-line , Julia allows statements of the form , and . Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. , because of the operator precedence.
- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
- Julia macros operate on parsed expressions, rather than the text of the program, which allows them to perform sophisticated transformations of Julia code. Macro names start with the character, and have both a function-like syntax, , and a statement-like syntax, . The forms are interchangable; the function-like form is particularly useful if the macro appears within another expression, and is often clearest. The statement-like form is often used to annotate blocks, as in the parallel construct: . Where the end of the macro construct may be unclear, use the function-like form.

- Julia now has an enumeration type, expressed using the macro For example:
- By convention, functions that modify their arguments have a at the end of the name, for example .
- In C++, by default, you have static dispatch, i.e. you need to annotate a function as virtual, in order to have dynamic dispatch. On the other hand, in Julia every method is ”virtual” (although it’s more general than that since methods are dispatched on every argument type, not only , using the most-specific-declaration rule).

Chapter 44

Unicode Input

The following table lists Unicode characters that can be entered via tab completion of LaTeX-like abbreviations in the Julia REPL (and in various other editing environments). You can also get information on how to type a symbol by entering it in the REPL help, i.e. by typing and then entering the symbol in the REPL (e.g., by copy-paste from somewhere you saw the symbol).

> **Warning**
>
> This table may appear to contain missing characters in the second column, or even show characters that are inconsistent with the characters as they are rendered in the Julia REPL. In these cases, users are strongly advised to check their choice of fonts in their browser and REPL environment, as there are known issues with glyphs in many fonts.

Part IV

Standard Library

Chapter 45

Essentials

45.1 Introduction

The Julia standard library contains a range of functions and macros appropriate for performing scientific and numerical computing, but is also as broad as those of many general purpose programming languages. Additional functionality is available from a growing collection of available packages. Functions are grouped by topic below.

Some general notes:

- Except for functions in built-in modules (, , and), all functions documented here are directly available for use in programs.
- To use module functions, use to import the module, and to use the functions.
- Alternatively, will import all exported functions into the current namespace.
- By convention, function names ending with an exclamation point () modify their arguments. Some functions have both modifying (e.g.,) and non-modifying () versions.

45.2 Getting Around

– Function.

Quit (or control-D at the prompt). The default exit code is zero, indicating that the processes completed successfully.

– Function.

Quit the program indicating that the processes completed successfully. This function calls (see).

– Function.

Register a zero-argument function to be called at process exit. hooks are called in last in first out (LIFO) order and run before object finalizers.

– Function.

Register a one-argument function to be called before the REPL interface is initialized in interactive sessions; this is useful to customize the interface. The argument of is the REPL object. This function should be called from within the initialization file.

– Function.

Determine whether Julia is running an interactive session.

– Function.

Print information about exported global variables in a module, optionally restricted to those matching .

The memory consumption estimate is an approximate lower bound on the size of the internal structure of the object.

– Function.

Compute the amount of memory used by all unique objects reachable from the argument.

Keyword Arguments

- : specifies the types of objects to exclude from the traversal.
- : specifies the types of objects to always charge the size of all of their fields, even if those fields would normally be excluded.

– Method.

Edit a file or directory optionally providing a line number to edit the file at. Returns to the prompt when you quit the editor. The editor can be changed by setting , or as an environment variable.

– Method.

Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit. The editor can be changed by setting , or as an environment variable.

– Macro.

Evaluates the arguments to the function or macro call, determines their types, and calls the function on the resulting expression.

– Method.

Show a file using the default pager, optionally providing a starting line number. Returns to the prompt when you quit the pager.

– Method.

Show the definition of a function using the default pager, optionally specifying a tuple of types to indicate which method to see.

– Macro.

Evaluates the arguments to the function or macro call, determines their types, and calls the function on the resulting expression.

– Method.

Send a printed form of to the operating system clipboard ("copy").

– Method.

Return a string with the contents of the operating system clipboard ("paste").

– Function.

Force reloading of a package, even if it has been loaded before. This is intended for use during package development as code is modified.

– Function.

This function is part of the implementation of / , if a module is not already defined in . It can also be called directly to force reloading a module, regardless of whether it has been loaded before (for example, when interactively developing libraries).

Loads a source file, in the context of the module, on every active node, searching standard locations for files. is considered a top-level operation, so it sets the current path but does not use it to search for files (see help for). This function is typically used to load library code, and is implicitly called by to load packages.

When searching for files, first looks for package code under , then tries paths in the global array . is case-sensitive on all platforms, including those with case-insensitive filesystems like macOS and Windows.

– Function.

Creates a precompiled cache file for a module and all of its dependencies. This can be used to reduce package load times. Cache files are stored in , which defaults to . See Module initialization and precompilation for important notes.

– Function.

Specify whether the file calling this function is precompilable. If is , then throws an exception when the file is loaded by // *unless* the file is being precompiled, and in a module file it causes the module to be automatically precompiled when it is imported. Typically, should occur before the declaration in the file.

If a module or file is *not* safely precompilable, it should call in order to throw an error if Julia attempts to precompile it.

should *not* be used in a module unless all of its dependencies are also using . Failure to do so can result in a runtime error when loading the module.

– Function.

Evaluate the contents of the input source file into module . Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to will search relative to that path. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

– Function.

Like , except reads code from the given string rather than from a file.

– Function.

In a module, declare that the file specified by (relative or absolute) is a dependency for precompilation; that is, the module will need to be recompiled if this file changes.

This is only needed if your module depends on a file that is not used via . It has no effect outside of compilation.

– Function.

Search through all documentation for a string, ignoring case.

– Method.

Returns the method of (a object) that would be called for arguments of the given .

If is an abstract type, then the method that would be called by is returned.

– Method.

Return the module in which the binding for the variable referenced by in module was created.

– Macro.

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns the object for the method that would be called for those arguments. Applied to a variable, it returns the module in which the variable was bound. It calls out to the function.

– Function.

Returns the method table for .

If is specified, returns an array of methods whose types match.

– Function.

Return an array of methods with an argument of type .

The optional second argument restricts the search to a particular module or function (the default is all modules, starting from Main).

If optional is , also return arguments with a parent type of , excluding type .

– Macro.

Show an expression and result, returning the result.

– Function.

Print information about the version of Julia in use. The output is controlled with boolean keyword arguments:

- : print information about installed packages
- : print all additional information

– Function.

Replace the top-level module () with a new one, providing a clean workspace. The previous module is made available as . A previously-loaded package can be accessed using a statement such as .

This function should only be used interactively.

– Keyword.

A variable referring to the last computed value, automatically set at the interactive prompt.

45.3 All Objects

– Function.

Determine whether and are identical, in the sense that no program could distinguish them. Compares mutable objects by address in memory, and compares immutable objects (such as numbers) by contents at the bit level. This function is sometimes called .

Examples

– Function.

Determine whether is of the given . Can also be used as an infix operator, e.g. .

– Method.

Similar to , except treats all floating-point values as equal to each other, and treats as unequal to . The default implementation of calls , so if you have a type that doesn't have these floating-point subtleties then you probably only need to define .

is the comparison function used by hash tables (). must imply that .

This typically means that if you define your own function then you must define a corresponding (and vice versa). Collections typically implement by calling recursively on all contents.

Scalar types generally do not need to implement separate from , unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on , , and).

Examples

– Method.

Similar to , except treats all floating-point values as equal to each other, and treats as unequal to . The default implementation of calls , so if you have a type that doesn't have these floating-point subtleties then you probably only need to define .

is the comparison function used by hash tables (). must imply that .

This typically means that if you define your own function then you must define a corresponding (and vice versa). Collections typically implement by calling recursively on all contents.

Scalar types generally do not need to implement separate from , unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on , , and).

Examples

If neither nor is null, compare them according to their values (i.e.). Else, return if both arguments are null, and if one is null but not the other: nulls are considered equal.

Examples

– Function.

Test whether is less than , according to a canonical total order. Values that are normally unordered, such as , are ordered in an arbitrary but consistent fashion. This is the default comparison used by . Non-numeric types with a canonical total order should implement this function. Numeric types only need to implement it if they have special values such as .

– Method.

If neither nor is null, compare them according to their values (i.e.). Else, return if only is null, and otherwise: nulls are always considered greater than non-nulls, but not greater than another null.

Examples

– Function.

Return if is , otherwise return . This differs from or in that it is an ordinary function, so all the arguments are evaluated first. In some cases, using instead of an statement can eliminate the branch in generated code and provide higher performance in tight loops.

Examples

– Function.

Compare and lexicographically and return -1, 0, or 1 depending on whether is less than, equal to, or greater than , respectively. This function should be defined for lexicographically comparable types, and will call by default.

Examples

– Function.

Determine whether is lexicographically less than .

Examples

– Function.

Get the concrete type of .

– Function.

Construct a tuple of the given objects.

Examples

– Function.

Create a tuple of length , computing each element as , where is the index of the element.

– Function.

Get a hash value for based on object identity. if .

– Function.

Compute an integer hash code such that implies . The optional second argument is a hash code to be mixed with the result.

New types should implement the 2-argument form, typically by calling the 2-argument method recursively in order to mix hashes of the contents with each other (and with). Typically, any type that implements should also implement its own (hence) to guarantee the property mentioned above.

– Function.

Register a function to be called when there are no program-accessible references to . The type of must be a , otherwise the behavior of this function is unpredictable.

– Function.

Immediately run finalizers registered for object .

– Function.

Create a shallow copy of : the outer structure is copied, but not all internal values. For example, copying an array produces a new array with identically-same elements as the original.

– Function.

Create a deep copy of : everything is copied recursively, resulting in a fully independent object. For example, deep-copying an array produces a new array whose elements are deep copies of the original elements. Calling on an object should generally have the same effect as serializing and then deserializing it.

As a special case, functions can only be actually deep-copied if they are anonymous, otherwise they are just copied. The difference is only relevant in the case of closures, i.e. functions which may contain hidden internal references.

While it isn't normally necessary, user-defined types can override the default behavior by defining a specialized version of the function (which shouldn't otherwise be used), where is the type to be specialized for, and keeps track of objects copied so far within the recursion. Within the definition, should be used in place of , and the variable should be updated as appropriate before returning.

– Function.

Tests whether an assignable location is defined. The arguments can be a module and a symbol or a composite object and field name (as a symbol) or index.

– Macro.

Tests whether variable is defined in the current scope.

Examples

– Function.

Convert to a value of type .

If is an type, an will be raised if is not representable by , for example if is not integer-valued, or is outside the range supported by .

Examples

If is a or type, then it will return the closest value to representable by .

If is a collection type and a collection, the result of may alias .

Similarly, if is a composite type and a related instance, the result of may alias part or all of .

– Function.

Convert all arguments to their common promotion type (if any), and return them all (as a tuple).

Examples

– Function.

Convert to the type of ().

– Function.

If is a type, return a ”larger” type (for numeric types, this will be a type with at least as much range and precision as the argument, and usually more). Otherwise is converted to .

Examples

– Function.

The identity function. Returns its argument.

Examples

45.4 Types

– Function.

Return the supertype of DataType .

– Function.

Subtype operator: returns if and only if all values of type are also of type .

– Function.

Supertype operator, equivalent to .

– Function.

Return a list of immediate subtypes of DataType . Note that all currently loaded subtypes are included, including those not visible in the current module.

Examples

– Function.

The lowest value representable by the given (real) numeric DataType .

Examples

– Function.

The highest value representable by the given (real) numeric .

– Function.

The smallest in absolute value non-subnormal value representable by the given floating-point DataType .

– Function.

The highest finite value representable by the given floating-point DataType .

Examples

– Function.

The largest integer losslessly representable by the given floating-point DataType .

The largest integer losslessly representable by the given floating-point DataType that also does not exceed the maximum integer representable by the integer DataType .

– Method.

Size, in bytes, of the canonical binary representation of the given DataType , if any.

Examples

If does not have a specific size, an error is thrown.

– Method.

Returns the *machine epsilon* of the floating point type (by default). This is defined as the gap between 1 and the next largest value representable by , and is equivalent to .

– Method.

Returns the *unit in last place* (ulp) of . This is the distance between consecutive representable floating point values at . In most cases, if the distance on either side of is different, then the larger of the two is taken, that is

The exceptions to this rule are the smallest and largest finite values (e.g. and for), which round to the smaller of the values.

The rationale for this behavior is that bounds the floating point rounding error. Under the default rounding mode, if y is a real number and x is the nearest floating point number to y, then

$$|y - x| \leq \mathrm{eps}(x)/2.$$

– Function.

Determine a type big enough to hold values of each argument type without loss, whenever possible. In some cases, where no type exists to which both types can be promoted losslessly, some loss is tolerated; for example, returns even though strictly, not all values can be represented exactly as values.

– Function.

Specifies what type should be used by when given values of types and . This function should not be called directly, but should have definitions added to it for new types as appropriate.

– Function.

Extract a named field from a of composite type. The syntax calls .

Examples

– Function.

Assign to a named field in of composite type. The syntax calls .

– Function.

The byte offset of field of a type relative to the data start. For example, we could use it in the following manner to summarize information about a struct:

– Function.

Determine the declared type of a field (specified by name or index) in a composite DataType .

Examples

– Function.

Return iff value is immutable. See Mutable Composite Types for a discussion of immutability. Note that this function works on values, so if you give it a type, it will tell you that a value of is mutable.

Examples

– Function.

Return if is a ”plain data” type, meaning it is immutable and contains no references to other values. Typical examples are numeric types such as , , and .

Examples

– Function.

Determine whether ’s only subtypes are itself and . This means is a concrete type that can have instances.

Examples

– Function.

Compute a type that contains both and .

– Function.

Compute a type that contains the intersection of and . Usually this will be the smallest such type or one close to it.

– Type.

Return , which contains no run-time data. Types like this can be used to pass the information between functions through the value , which must be an value. The intent of this construct is to be able to dispatch on constants directly (at compile time) without having to test the value of the constant at run time.

Examples

```

```

– Macro.

Create an subtype with name and enum member values of and with optional assigned values of and , respectively. can be used just like other types and enum member values as regular values, such as

Examples

```

```

, which defaults to , must be a primitive subtype of . Member values can be converted between the enum type and . and perform these conversions automatically.

– Function.

Return a collection of all instances of the given type, if applicable. Mostly used for enumerated types (see).

Example

```

```

45.5 Generic Functions

– Type.

Abstract type of all functions.

– Function.

Determine whether the given generic function has a method matching the given of argument types with the upper bound of world age given by .

Examples

– Function.

Determine whether the given generic function has a method applicable to the given arguments.

Examples

– Function.

Invoke a method for the given generic function matching the specified types, on the specified arguments. The arguments must be compatible with the specified types. This allows invoking a method other than the most specific matching method, which is useful when the behavior of a more general definition is explicitly needed (often as part of the implementation of a more specific method of the same function).

– Function.

Calls , but guarantees that the most recent method of will be executed. This is useful in specialized circumstances, e.g. long-running event loops or callback functions that may call obsolete versions of a function . (The drawback is that is somewhat slower than calling directly, and the type of the result cannot be inferred by the compiler.)

– Function.

Applies a function to the preceding argument. This allows for easy function chaining.

Examples

– Function.

Compose functions: i.e. means . The symbol can be entered in the Julia REPL (and most editors, appropriately configured) by typing .

Examples

45.6 Syntax

– Function.

Evaluate an expression in the given module and return the result. Every (except those defined with) has its own 1-argument definition of , which evaluates expressions in that module.

– Macro.

Evaluate an expression with values interpolated into it using . If two arguments are provided, the first is the module to evaluate in.

– Function.

Load the file using , evaluate all expressions, and return the value of the last one.

– Function.

Only valid in the context of an returned from a macro. Prevents the macro hygiene pass from turning embedded variables into gensym variables. See the Macros section of the Metaprogramming chapter of the manual for more details and examples.

– Macro.

Eliminates array bounds checking within expressions.

In the example below the bound check of array A is skipped to improve performance.

Warning

Using may return incorrect results/crashes/corruption for out-of-bounds indices. The user is responsible for checking it manually.

– Macro.

Give a hint to the compiler that this function is worth inlining.

Small functions typically do not need the annotation, as the compiler does it automatically. By using on bigger functions, an extra nudge can be given to the compiler to inline it. This is shown in the following example:

– Macro.

Prevents the compiler from inlining a function.

Small functions are typically inlined automatically. By using on small functions, auto-inlining can be prevented. This is shown in the following example:

– Macro.

Applied to a function argument name, hints to the compiler that the method should not be specialized for different types of that argument. This is only a hint for avoiding excess code generation. Can be applied to an argument within a formal argument list, or in the function body. When applied to an argument, the macro must wrap the entire argument expression. When used in a function body, the macro must occur in statement position and before any code.

– Function.

Generates a symbol which will not conflict with other variable names.

– Macro.

Generates a gensym symbol for a variable. For example, is transformed into .

– Macro.

Tells the compiler to apply the polyhedral optimizer Polly to a function.

– Method.

Parse the expression string and return an expression (which could later be passed to eval for execution). is the index of the first character to start parsing. If is (default), will try to consume as much input as it can; otherwise, it will stop as soon as it has parsed a valid expression. Incomplete but otherwise syntactically valid expressions will return . If is (default), syntax errors other than incomplete expressions will raise an error. If is , will return an expression that will raise an error upon evaluation.

– Method.

Parse the expression string greedily, returning a single expression. An error is thrown if there are additional characters after the first expression. If is (default), syntax errors will raise an error; otherwise, will return an expression that will raise an error upon evaluation.

45.7 Nullables

– Type.

Wrap value in an object of type , which indicates whether a value is present. yields a non-empty wrapper and yields an empty instance of a wrapper that might contain a value of type .

yields with stored in the result's field.

Examples

– Method.

Attempt to access the value of . Returns the value if it is present; otherwise, returns if provided, or throws a if not.

Examples

– Function.

Return whether or not is null for ; return for all other .

Examples

– Function.

Return the value of for ; return for all other .

This method does not check whether or not is null before attempting to access the value of for (hence ”unsafe”).

Examples

45.8 System

– Function.

Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.

– Function.

Run a command object asynchronously, returning the resulting object.

– Constant.

Used in a stream redirect to discard all data written to it. Essentially equivalent to /dev/null on Unix or NUL on Windows. Usage:

– Function.

Run a command object, constructed with backticks, and tell whether it was successful (exited with a code of 0). An exception is raised if the process cannot be started.

– Function.

Determine whether a process is currently running.

– Function.

Determine whether a process has exited.

– Method.

Send a signal to a process. The default is to terminate the process.

– Function.

Set the process title. No-op on some operating systems.

– Function.

Get the process title. On some systems, will always return an empty string.

– Function.

Starts running a command asynchronously, and returns a tuple (stdout,stdin,process) of the output stream and input stream of the process, and the process object itself.

– Function.

Mark a command object so that running it will not throw an error if the result code is non-zero.

– Function.

Mark a command object so that it will be run in a new process group, allowing it to outlive the julia process, and not have Ctrl-C interrupts passed to it.

– Type.

Construct a new object, representing an external program and arguments, from , while changing the settings of the optional keyword arguments:

- : If (defaults to), then the will not throw an error if the return code is nonzero.
- : If (defaults to), then the will be run in a new process group, allowing it to outlive the process and not have Ctrl-C passed to it.
- : If (defaults to), then on Windows the will send a command-line string to the process with no quoting or escaping of arguments, even arguments containing spaces. (On Windows, arguments are sent to a program as a single "command-line" string, and programs are responsible for parsing it into arguments. By default, empty arguments and arguments with spaces or tabs are quoted with double quotes in the command line, and or are preceded by backslashes. is useful for launching programs that parse their command line in nonstandard ways.) Has no effect on non-Windows systems.
- : If (defaults to), then on Windows no new console window is displayed when the is executed. This has no effect if a console is already open or on non-Windows systems.

- : Set environment variables to use when running the . is either a dictionary mapping strings to strings, an array of strings of the form , an array or tuple of pairs, or . In order to modify (rather than replace) the existing environment, create by and then set as desired.
- : Specify a working directory for the command (instead of the current directory).

For any keywords that are not specified, the current settings from are used. Normally, to create a object in the first place, one uses backticks, e.g.

– Function.

Set environment variables to use when running the given . is either a dictionary mapping strings to strings, an array of strings of the form , or zero or more pair arguments. In order to modify (rather than replace) the existing environment, create by and then setting as desired, or use .

The keyword argument can be used to specify a working directory for the command.

– Function.

Execute in an environment that is temporarily modified (not replaced as in) by zero or more arguments . is generally used via the syntax. A value of can be used to temporarily unset an environment variable (if it is set). When returns, the original environment has been restored.

– Method.

Create a pipeline from a data source to a destination. The source and destination can be commands, I/O streams, strings, or results of other calls. At least one argument must be a command. Strings refer to filenames. When called with more than two arguments, they are chained together from left to right. For example is equivalent to . This provides a more concise way to specify multi-stage pipelines.

Examples:

– Method.

Redirect I/O to or from the given . Keyword arguments specify which of the command's streams should be redirected. controls whether file output appends to the file. This is a more general version of the 2-argument function. is equivalent to when is a command, and to when is another kind of data source.

Examples:

– Function.

Get the local machine's host name.

– Function.

Get the IP address of the local machine.

– Function.

Get Julia's process ID.

– Method.

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

– Function.

Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

– Function.

Set a timer to be read by the next call to or . The macro call can also be used to time evaluation.

– Function.

Print and return the time elapsed since the last . The macro call can also be used to time evaluation.

– Function.

Return, but do not print, the time elapsed since the last . The macro calls and also return evaluation time.

– Macro.

A macro to execute an expression, printing the time it took to execute, the number of allocations, and the total number of bytes its execution caused to be allocated, before returning the value of the expression.

See also , , , and .

– Macro.

This is a verbose version of the macro. It first prints the same information as , then any non-zero memory allocation counters, and then returns the value of the expression.

See also , , , and .

– Macro.

A macro to execute an expression, and return the value of the expression, elapsed time, total bytes allocated, garbage collection time, and an object with various memory allocation counters.

See also , , , and .

– Macro.

A macro to evaluate an expression, discarding the resulting value, instead returning the number of seconds it took to execute as a floating-point number.

See also , , , and .

– Macro.

A macro to evaluate an expression, discarding the resulting value, instead returning the total number of bytes allocated during evaluation of the expression. Note: the expression is evaluated inside a local function, instead of the current context, in order to eliminate the effects of compilation, however, there still may be some allocations due to JIT compilation. This also makes the results inconsistent with the macros, which do not try to adjust for the effects of compilation.

See also , , , and .

– Type.

A singleton of this type provides a hash table interface to environment variables.

– Constant.

Reference to the singleton , providing a dictionary interface to system environment variables.

– Function.

Predicate for testing if the OS provides a Unix-like interface. See documentation in Handling Operating System Variation.

– Function.

Predicate for testing if the OS is a derivative of Apple Macintosh OS X or Darwin. See documentation in Handling Operating System Variation.

– Function.

Predicate for testing if the OS is a derivative of Linux. See documentation in Handling Operating System Variation.

– Function.

Predicate for testing if the OS is a derivative of BSD. See documentation in Handling Operating System Variation.

– Function.

Predicate for testing if the OS is a derivative of Microsoft Windows NT. See documentation in Handling Operating System Variation.

– Function.

Returns the version number for the Windows NT Kernel as a , i.e. , or if this is not running on Windows.

– Macro.

Partially evaluate an expression at parse time.

For example, will evaluate and insert either or into the expression. This is useful in cases where a construct would be invalid on other platforms, such as a to a non-existent function. and are also valid syntax.

45.9 Errors

– Function.

Raise an with the given message.

– Function.

Throw an object as an exception.

– Function.

Throw an object without changing the current exception backtrace. The default argument is the current exception (if called within a block).

– Function.

Get a backtrace object for the current program point.

– Function.

Get the backtrace of the current exception, for use within blocks.

– Function.

Throw an if is . Also available as the macro .

– Macro.

Throw an if is . Preferred syntax for writing assertions. Message is optionally displayed upon assertion failure.

Examples

– Type.

The parameters to a function call do not match a valid signature. Argument is a descriptive error string.

– Type.

The asserted condition did not evaluate to . Optional argument is a descriptive error string.

– Type.

An indexing operation into an array, , tried to access an out-of-bounds element at index .

Examples

– Type.

The objects called do not have matching dimensionality. Optional argument is a descriptive error string.

– Type.

Integer division was attempted with a denominator value of 0.

Examples

– Type.

The argument to a function or constructor is outside the valid domain.

Examples

– Type.

No more data was available to read from a file or stream.

– Type.

Generic error type. The error message, in the field, may provide more specific details.

– Type.

Cannot exactly convert to type in a method of function .

Examples

– Type.

The process was stopped by a terminal interrupt (CTRL+C).

– Type.

An indexing operation into an () or like object tried to access or delete a non-existent element.

– Type.

An error occurred while ing, ing, or a file. The error specifics should be available in the field.

– Type.

A method with the required type signature does not exist in the given generic function. Alternatively, there is no unique most-specific method.

– Type.

An attempted access to a with no defined value.

Examples

– Type.

An operation allocated too much memory for either the system or the garbage collector to handle properly.

– Type.

An operation tried to write to memory that is read-only.

– Type.

The result of an expression is too large for the specified type and will cause a wraparound.

– Type.

The expression passed to the function could not be interpreted as a valid Julia expression.

– Type.

After a client Julia process has exited, further attempts to reference the dead child will throw this exception.

– Type.

The function call grew beyond the size of the call stack. This usually happens when a call recurses infinitely.

– Type.

A system call failed with an error code (in the global variable).

– Type.

A type assertion failure, or calling an intrinsic function with an incorrect argument type.

– Type.

The item or field is not defined for the given object.

– Type.

A symbol in the current scope is not defined.

– Type.

An error occurred when running a module's function. The actual error thrown is available in the field.

– Function.

Returns an anonymous function that calls function . If an exception arises, is repeatedly called again, each time returns , after waiting the number of seconds specified in . should input 's current state and the .

Examples

– Type.

A iterator of length whose elements exponentially increase at a rate in the interval * (1 ±). The first element is and all elements are clamped to .

45.10 Events

– Method.

Create a timer to call the given function. The is passed one argument, the timer object itself. The callback will be invoked after the specified initial , and then repeating with the given interval. If is , the timer is only triggered once. Times are in seconds. A timer is stopped and has its resources freed by calling on it.

– Type.

Create a timer that wakes up tasks waiting for it (by calling on the timer object) at a specified interval. Times are in seconds. Waiting tasks are woken with an error when the timer is closed (by . Use to check whether a timer is still active.

– Type.

Create a async condition that wakes up tasks waiting for it (by calling on the object) when notified from C by a call to . Waiting tasks are woken with an error when the object is closed (by . Use to check whether it is still active.

– Method.

Create a async condition that calls the given function. The is passed one argument, the async condition object itself.

45.11 Reflection

– Function.

Get the name of a as a .

Examples

– Function.

Get a module's enclosing . is its own parent, as is after .

Examples

– Macro.

Get the of the toplevel eval, which is the code is currently being read from.

– Function.

Get the fully-qualified name of a module as a tuple of symbols. For example,

Examples

– Function.

Get an array of the names exported by a , excluding deprecated names. If is true, then the list also includes non-exported names defined in the module, deprecated names, and compiler-generated names. If is true, then names explicitly imported from other modules are also included.

As a special case, all names defined in are considered ”exported”, since it is not idiomatic to explicitly export names from .

– Function.

Get the number of fields in the given object.

– Function.

Get an array of the fields of a .

Examples

– Function.

Get the name of field of a .

Examples

– Function.

Get the number of fields that an instance of the given type would have. An error is thrown if the type is too abstract to determine this.

– Function.

Determine the module containing the definition of a .

Examples

– Function.

Get the name of a (potentially UnionAll-wrapped) (without its parent module) as a symbol.

Examples

– Function.

Determine whether a global is declared in a given .

– Function.

Get the name of a generic as a symbol, or .

– Method.

Determine the module containing the (first) definition of a generic function.

– Method.

Determine the module containing a given definition of a generic function.

– Method.

Returns a tuple giving the location of a generic definition.

– Method.

Returns a tuple giving the location of a definition.

– Macro.

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns a tuple giving the location for the method that would be called for those arguments. It calls out to the function.

45.12 Internals

– Function.

Perform garbage collection. This should not generally be used.

– Function.

Control whether garbage collection is enabled using a boolean argument (for enabled, for disabled). Returns previous GC state. Disabling garbage collection should be used only with extreme caution, as it can cause memory use to grow without bound.

– Function.

Takes the expression and returns an equivalent expression with all macros removed (expanded) for executing in module . The keyword controls whether deeper levels of nested macros are also expanded. This is demonstrated in the example below:

– Macro.

Return equivalent expression with all macros removed (expanded).

There are differences between and .

- While takes a keyword argument ,

is always recursive. For a non recursive macro version, see .

- While has an explicit argument, always

expands with respect to the module in which it is called. This is best seen in the following example:

With the expression expands where appears in the code (module in the example). With the expression expands in the module given as the first argument.

– Macro.

Non recursive version of .

– Function.

Takes the expression and returns an equivalent expression in lowered form for executing in module . See also .

– Function.

Returns an array of lowered ASTs for the methods matching the given generic function and type signature.

– Macro.

Evaluates the arguments to the function or macro call, determines their types, and calls on the resulting expression.

– Function.

Returns an array of lowered and type-inferred ASTs for the methods matching the given generic function and type signature. The keyword argument controls whether additional optimizations, such as inlining, are also applied.

– Macro.

Evaluates the arguments to the function or macro call, determines their types, and calls on the resulting expression.

– Function.

Prints lowered and type-inferred ASTs for the methods matching the given generic function and type signature to which defaults to . The ASTs are annotated in such a way as to cause ”non-leaf” types to be emphasized (if color is available, displayed in red). This serves as a warning of potential type instability. Not all non-leaf types are particularly problematic for performance, so the results need to be used judiciously. See for more information.

– Macro.

Evaluates the arguments to the function or macro call, determines their types, and calls on the resulting expression.

– Function.

Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to which defaults to .

All metadata and dbg.* calls are removed from the printed bitcode. Use code_llvm_raw for the full IR.

– Macro.

Evaluates the arguments to the function or macro call, determines their types, and calls on the resulting expression.

– Function.

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to which defaults to . Switch assembly syntax using symbol parameter set to for AT&T syntax or for Intel syntax. Output is AT&T syntax by default.

– Macro.

Evaluates the arguments to the function or macro call, determines their types, and calls on the resulting expression.

– Function.

Compile the given function for the argument tuple (of types) , but do not execute it.

Chapter 46

Collections and Data Structures

46.1 Iteration

Sequential iteration is implemented by the methods , , and . The general loop:

is translated into:

The object may be anything, and should be chosen appropriately for each iterable type. See the manual section on the iteration interface for more details about defining a custom iterable type.

– Function.

Get initial iteration state for an iterable object.

Examples

– Function.

Test whether we are done iterating.

Examples

– Function.

For a given iterable object and iteration state, return the current item and the next iteration state.

Examples

– Function.

Given the type of an iterator, returns one of the following values:

- if the length (number of elements) cannot be determined in advance.
- if there is a fixed, finite length.
- if there is a known length plus a notion of multidimensional shape (as for an array). In this case the function is valid for the iterator.
- if the iterator yields values forever.

The default value (for iterators that do not define this function) is . This means that most iterators are assumed to implement .

This trait is generally used to select between algorithms that pre-allocate space for their result, and algorithms that resize their result incrementally.

– Function.

Given the type of an iterator, returns one of the following values:

- if the type of elements yielded by the iterator is not known in advance.
- if the element type is known, and would return a meaningful value.

is the default, since iterators are assumed to implement .

This trait is generally used to select between algorithms that pre-allocate a specific type of result, and algorithms that pick a result type based on the types of yielded values.

Fully implemented by:

-
-
-
-
-
-
-
-
-
-
-

46.2 General Collections

– Function.

Determine whether a collection is empty (has no elements).

Examples

– Function.

Remove all elements from a .

– Method.

Return the number of elements in the collection.

Use to get the last valid index of an indexable collection.

Examples

Fully implemented by:

-
-
-
-

-
-
-
-
-
-
-

46.3 Iterable Collections

– Function.

Determine whether an item is in the given collection, in the sense that it is to one of the values generated by iterating over the collection. Some collections need a slightly different definition; for example s check whether the item to one of the elements. s look for pairs, and the key is compared using . To test for the presence of a key in a dictionary, use or .

– Function.

Determine the type of the elements generated by iterating a collection of the given . For associative collection types, this will be a . The definition is provided for convenience so that instances can be passed instead of types. However the form that accepts a type argument should be defined for new types.

Examples

– Function.

Returns a vector containing the highest index in for each value in that is a member of . The output vector contains 0 wherever is not a member of .

Examples

– Function.

Returns the indices of elements in collection that appear in collection .

Examples

– Function.

Return an array containing only the unique elements of collection , as determined by , in the order that the first of each set of equivalent elements originally appears.

Examples

Returns an array containing one value from for each unique value produced by applied to elements of .

Examples

Return unique regions of along dimension .

Examples

– Function.

Remove duplicate items as determined by , then return the modified . will return the elements of in the order that they occur. If you do not care about the order of the returned data, then calling will be much more efficient as long as the elements of can be sorted.

Examples

– Function.

Return if all values from are distinct when compared with .

Examples

– Method.

Reduce the given collection with the given binary operator . must be a neutral element for that will be returned for empty collections. It is unspecified whether is used for non-empty collections.

Reductions for certain commonly-used operators have special implementations which should be used instead: , , , , , .

The associativity of the reduction is implementation dependent. This means that you can't use non-associative operations like because it is undefined whether should be evaluated as or . Use or instead for guaranteed left or right associativity.

Some operations accumulate error, and parallelism will also be easier if the reduction can be executed in groups. Future versions of Julia might change the algorithm. Note that the elements are not reordered if you use an ordered collection.

Examples

– Method.

Like . This cannot be used with empty collections, except for some special cases (e.g. when is one of , , , , ,) when Julia can determine the neutral element of .

– Method.

Like , but with guaranteed left associativity. will be used exactly once.

– Method.

Like , but using the first element of as . In general, this cannot be used with empty collections (see).

– Method.

Like , but with guaranteed right associativity. will be used exactly once.

– Method.

Like , but using the last element of as . In general, this cannot be used with empty collections (see).

– Method.

Returns the largest element in a collection.

– Method.

Compute the maximum value of an array over the given dimensions. See also the function to take the maximum of two or more arguments, which can be applied elementwise to arrays via .

– Function.

Compute the maximum value of over the singleton dimensions of , and write results to .

Examples

– Method.

Returns the smallest element in a collection.

– Method.

Compute the minimum value of an array over the given dimensions. See also the function to take the minimum of two or more arguments, which can be applied elementwise to arrays via .

Examples

```

```

– Function.

```

```

Compute the minimum value of over the singleton dimensions of , and write results to .

Examples

```

```

– Method.

```

```

Compute both the minimum and maximum element in a single pass, and return them as a 2-tuple.

```

```

– Method.

Compute the minimum and maximum elements of an array over the given dimensions.

Examples

– Function.

Returns the index of the maximum element in a collection. If there are multiple maximal elements, then the first one will be returned. values are ignored, unless all elements are .

The collection must not be empty.

Examples

– Function.

Returns the index of the minimum element in a collection. If there are multiple minimal elements, then the first one will be returned. values are ignored, unless all elements are .

The collection must not be empty.

Examples

– Method.

Returns the maximum element of the collection and its index. If there are multiple maximal elements, then the first one will be returned. values are ignored, unless all elements are .

The collection must not be empty.

Examples

– Method.

For an array input, returns the value and index of the maximum over the given region.

Examples

– Method.

Returns the minimum element of the collection and its index. If there are multiple minimal elements, then the first one will be returned. values are ignored, unless all elements are .

The collection must not be empty.

Examples

– Method.

For an array input, returns the value and index of the minimum over the given region.

Examples

– Function.

Find the maximum of and the corresponding linear index along singleton dimensions of and , and store the results in and .

– Function.

Find the minimum of and the corresponding linear index along singleton dimensions of and , and store the results in and .

– Function.

Sum the results of calling function on each element of .

Returns the sum of all elements in a collection.

Sum elements of an array over the given dimensions.

Examples

– Function.

Sum elements of over the singleton dimensions of , and write results to .

Examples

– Function.

Returns the product of applied to each element of .

Returns the product of all elements of a collection.

Multiply elements of an array over the given dimensions.

Examples

– Function.

Multiply elements of over the singleton dimensions of , and write results to .

Examples

– Method.

Test whether any elements of a boolean collection are , returning as soon as the first value in is encountered (short-circuiting).

– Method.

Test whether any values along the given dimensions of an array are .

Examples

– Function.

Test whether any values in along the singleton dimensions of are , and write results to .

Examples

– Method.

Test whether all elements of a boolean collection are , returning as soon as the first value in is encountered (short-circuiting).

– Method.

Test whether all values along the given dimensions of an array are .

Examples

– Function.

Test whether all values in along the singleton dimensions of are , and write results to .

Examples

– Function.

Count the number of elements in for which predicate returns . If is omitted, counts the number of elements in (which should be a collection of boolean values).

– Method.

Determine whether predicate returns for any elements of , returning as soon as the first item in for which returns is encountered (short-circuiting).

– Method.

Determine whether predicate returns for all elements of , returning as soon as the first item in for which returns is encountered (short-circuiting).

– Function.

Call function on each element of iterable . For multiple iterable arguments, is called elementwise. should be used instead of when the results of are not needed, for example in .

Examples

– Function.

Transform collection by applying to each element. For multiple collection arguments, apply elementwise.

Examples

Return applied to the value of if it has one, as a . If is null, then return a null value of type . is guaranteed to be either or a concrete type. Whichever of these is chosen is an implementation detail, but typically the choice that maximizes performance would be used. If has a value, then the return type is guaranteed to be of type .

Examples

– Function.

Like , but stores the result in rather than a new collection. must be at least as large as the first collection.

Examples

– Method.

Apply function to each element in , and then reduce the result using the binary function . must be a neutral element for that will be returned for empty collections. It is unspecified whether is used for non-empty collections.

 is functionally equivalent to calling , but will in general execute faster since no intermediate collection needs to be created. See documentation for and .

The associativity of the reduction is implementation-dependent. Additionally, some implementations may reuse the return value of for elements that appear multiple times in . Use or instead for guaranteed left or right associativity and invocation of for every value.

– Method.

Like . In general, this cannot be used with empty collections (see).

– Method.

Like , but with guaranteed left associativity, as in . will be used exactly once.

– Method.

Like , but using the first element of as . In general, this cannot be used with empty collections (see).

– Method.

Like , but with guaranteed right associativity, as in . will be used exactly once.

– Method.

Like , but using the first element of as . In general, this cannot be used with empty collections (see).

– Function.

Get the first element of an iterable collection. Returns the start point of a even if it is empty.

Examples

– Function.

Get the last element of an ordered collection, if it can be computed in O(1) time. This is accomplished by calling to get the last index. Returns the end point of a even if it is empty.

Examples

– Function.

Get the step size of a object.

– Method.

Return an of all items in a collection or iterator. For associative collections, returns . If the argument is array-like or is an iterator with the trait, the result will have the same shape and number of dimensions as the argument.

Examples

– Method.

Return an with the given element type of all items in a collection or iterable. The result has the same shape and number of dimensions as .

Examples

– Method.

Determine whether every element of is also in , using .

Examples

– Function.

Return a copy of , removing elements for which is . The function is passed one argument.

Examples

Return a copy of , removing elements for which is . The function is passed two arguments (key and value).

Examples

Return null if either is null or is false, and otherwise.

Examples

– Function.

Update , removing elements for which is . The function is passed one argument.

Examples

Update , removing elements for which is . The function is passed two arguments (key and value).

Example

46.4 Indexable Collections

– Method.

Retrieve the value(s) stored at the given key or index within a collection. The syntax is converted by the compiler to .

Examples

– Method.

Store the given value at the given key or index within a collection. The syntax is converted by the compiler to .

– Function.

Returns the last index of the collection.

Examples

Fully implemented by:

-
-
-
-

Partially implemented by:

-
-
-
-
-
-
-

46.5 Associative Collections

is the standard associative collection. Its implementation uses as the hashing function for the key, and to determine equality. Define these two functions for custom types to override how they are stored in a hash table.

is a special hash table where the keys are always object identities.

is a hash table implementation where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table.

s can be created by passing pair objects constructed with to a constructor: . This call will attempt to infer type information from the keys and values (i.e. this example creates a). To explicitly specify types use the syntax . For example, .

Associative collections may also be created with generators. For example, .

Given a dictionary , the syntax returns the value of key (if it exists) or throws an error, and stores the key-value pair in (replacing any existing value for the key). Multiple arguments to are converted to tuples; for example, the syntax is equivalent to , i.e. it refers to the value keyed by the tuple .

– Type.

constructs a hash table with keys of type and values of type .

Given a single iterable argument, constructs a whose key-value pairs are taken from 2-tuples generated by the argument.

Alternatively, a sequence of pair arguments may be passed.

– Type.

constructs a hash table where the keys are (always) object identities. Unlike it is not parameterized on its key and value type and thus its is always .

See for further help.

– Type.

constructs a hash table where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table.

See for further help.

– Function.

Determine whether a collection has a mapping for a given key.

– Method.

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Examples

– Function.

Return the value stored for the given key, or if no mapping for the key is present, return . Use to also store the default value in the dictionary.

This is intended to be called using block syntax

– Method.

Return the value stored for the given key, or if no mapping for the key is present, store , and return .

Examples

– Method.

Return the value stored for the given key, or if no mapping for the key is present, store , and return .

This is intended to be called using block syntax:

– Function.

Return the key matching argument if one exists in , otherwise return .

– Function.

Delete the mapping for the given key in a collection, and return the collection.

Examples

– Method.

Delete and return the mapping for if it exists in , otherwise return , or throw an error if is not specified.

Examples

– Function.

Return an iterator over all keys in a collection. returns an array of keys. Since the keys are stored internally in a hash table, the order in which they are returned may vary. But and both iterate and return the elements in the same order.

Examples

```
```

– Function.

```
```

Return an iterator over all values in a collection. returns an array of values. Since the values are stored internally in a hash table, the order in which they are returned may vary. But and both iterate and return the elements in the same order.

Examples

```
```

– Function.

```
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. If the same key is present in another collection, the value for that key will be the value it has in the last collection listed.

Examples

```
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. Values with the same key will be combined using the combiner function.

Examples

– Method.

Update collection with pairs from the other collections. See also .

Examples

– Method.

Update collection with pairs from the other collections. Values with the same key will be combined using the combiner function.

Examples

– Function.

Suggest that collection reserve capacity for at least elements. This can improve performance.

– Function.

Get the key type of an associative collection type. Behaves similarly to .

Examples

– Function.

Get the value type of an associative collection type. Behaves similarly to .

Examples

Fully implemented by:

-
-
-

Partially implemented by:

-
-
-
-
-

46.6 Set-Like Collections

– Type.

Construct a of the values generated by the given iterable object, or an empty set. Should be used instead of for sparse integer sets, or for sets of arbitrary objects.

– Type.

Construct a sorted set of positive s generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. Only s greater than 0 can be stored. If the set will be sparse (for example holding a few very large integers), use instead.

– Function.

Construct the union of two or more sets. Maintains order with arrays.

Examples

– Function.

Union each element of into set in-place.

Examples

– Function.

Construct the intersection of two or more sets. Maintains order and multiplicity of the first argument for arrays and ranges.

Examples

– Function.

Construct the set of elements in but not . Maintains order with arrays. Note that both arguments must be collections, and both will be iterated over. In particular, where is a potential member of , will not work in general.

Examples

– Function.

Remove each element of from set in-place.

Examples

– Function.

Construct the symmetric difference of elements in the passed in sets or arrays. Maintains order with arrays.

Examples

– Method.

The set is destructively modified to toggle the inclusion of integer .

– Method.

For each element in , destructively toggle its inclusion in set .

– Method.

For each element in , destructively toggle its inclusion in set .

– Function.

Intersects sets and and overwrites the set with the result. If needed, will be expanded to the size of .

– Function.

Determine whether every element of is also in , using .

Examples

Fully implemented by:

-
-

Partially implemented by:

-

46.7 Dequeues

– Function.

Insert one or more at the end of .

Examples

Use to add all the elements of another collection to . The result of the preceding example is equivalent to .

– Method.

Remove an item in and return it. If is an ordered container, the last item is returned.

Examples

– Function.

Insert one or more at the beginning of .

Examples

– Function.

Remove the first from .

Examples

– Function.

Insert an into at the given . is the index of in the resulting .

Examples

– Function.

Remove the item at the given and return the modified . Subsequent items are shifted to fill the resulting gap.

Examples

Remove the items at the indices given by , and return the modified . Subsequent items are shifted to fill the resulting gap.

 can be either an iterator or a collection of sorted and unique integer indices, or a boolean vector of the same length as with indicating entries to delete.

Examples

– Function.

Remove the item at the given index, and return the removed item. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed item.

Examples

To insert before an index without removing any items, use .

Remove items in the specified index range, and return a collection containing the removed items. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed items.

To insert before an index without removing any items, use .

Examples

– Function.

Resize to contain elements. If is smaller than the current collection length, the first elements will be retained. If is larger, the new elements are not guaranteed to be initialized.

Examples

– Function.

Add the elements of to the end of .

Examples

Use to add individual items to which are not already themselves in another collection. The result is of the preceding example is equivalent to .

– Function.

Insert the elements of to the beginning of .

Examples

Fully implemented by:

- (a.k.a. 1-dimensional)
- (a.k.a. 1-dimensional)

Chapter 47

Mathematics

47.1 Mathematical Operators

– Method.

Unary minus operator.

– Function.

Addition operator. calls this function with all arguments, i.e. .

– Method.

Subtraction operator.

– Method.

Multiplication operator. calls this function with all arguments, i.e. .

– Function.

Right division operator: multiplication of by the inverse of on the right. Gives floating-point results for integer arguments.

– Method.

Left division operator: multiplication of by the inverse of on the left. Gives floating-point results for integer arguments.

Examples

– Method.

Exponentiation operator. If is a matrix, computes matrix exponentiation.

If is an literal (e.g. in or in), the Julia code is transformed by the compiler to , to enable compile-time specialization on the value of the exponent. (As a default fallback we have , where usually unless has been defined in the calling namespace.)

– Function.

Computes without rounding the intermediate result . On some systems this is significantly more expensive than . is used to improve accuracy in certain algorithms. See .

– Function.

Combined multiply-add, computes allowing the add and multiply to be contracted with each other or ones from other and to form if the transformation can improve performance. The result can be different on different machines and can also be different on the same machine due to constant propagation or other optimizations. See .

Examples

– Method.

Return the multiplicative inverse of , such that or yields (the multiplicative identity) up to roundoff errors.

If is a number, this is essentially the same as , but for some types may be slightly more efficient.

Examples

– Function.

The quotient from Euclidean division. Computes , truncated to an integer.

Examples

– Function.

Largest integer less than or equal to .

Examples

– Function.

Smallest integer larger than or equal to .

Examples

– Function.

The reduction of modulo , or equivalently, the remainder of after floored division by , i.e.

if computed without intermediate rounding.

The result will have the same sign as , and magnitude less than (with some exceptions, see note below).

> **Note**
> When used with floating point values, the exact result may not be representable by the type, and so rounding error may occur. In particular, if the exact result is very close to , then it may be rounded to .

Find such that (mod n), where n is the number of integers representable in , and is an integer in . If can represent any integer (e.g.), then this operation corresponds to a conversion to .

– Function.

Remainder from Euclidean division, returning a value of the same sign as , and smaller in magnitude than . This value is always exact.

Examples

– Function.

Compute the remainder of after integer division by , with the quotient rounded according to the rounding mode . In other words, the quantity

without any intermediate rounding. This internally uses a high precision approximation of 2π, and so will give a more accurate result than

- if , then the result is in the interval $[-,]$. This will generally be the most accurate result.

- if , then the result is in the interval $[0, 2]$ if is positive,. or $[-2, 0]$ otherwise.
- if , then the result is in the interval $[0, 2]$.
- if , then the result is in the interval $[-2, 0]$.

Examples

```

```

– Function.

```

```

Modulus after division by , returning in the range $[0, 2)$.

This function computes a floating point representation of the modulus after division by numerically exact , and is therefore not exactly the same as , which would compute the modulus of relative to division by the floating-point number .

Examples

```

```

– Function.

```

```

The quotient and remainder from Euclidean division. Equivalent to or .

```

```

– Function.

```

```

The floored quotient and modulus after division. Equivalent to .

– Function.

```

```

Flooring division, returning a value consistent with

See also: .

Examples

– Function.

Modulus after flooring division, returning a value such that in the range $(0, y]$ for positive and in the range $[y, 0)$ for negative .

Examples

– Function.

Return .

See also: , .

– Function.

Divide two integers or rational numbers, giving a result.

– Function.

Approximate floating point number as a number with components of the given integer type. The result will differ from by no more than . If is not provided, it defaults to .

– Function.

Numerator of the rational representation of .

– Function.

Denominator of the rational representation of .

– Function.

Left bit shift operator, . For , the result is shifted left by bits, filling with s. This is equivalent to . For , this is equivalent to .

Examples

See also , .

Left bit shift operator, . For , the result is with elements shifted positions backwards, filling with values. If , elements are shifted forwards. Equivalent to .

Examples

– Function.

Right bit shift operator, . For , the result is shifted right by bits, where , filling with s if , s if , preserving the sign of . This is equivalent to . For , this is equivalent to .

Examples

See also , .

Right bit shift operator, . For , the result is with elements shifted positions forward, filling with values. If , elements are shifted backwards. Equivalent to .

Examples

– Function.

Unsigned right bit shift operator, . For , the result is shifted right by bits, where , filling with s. For , this is equivalent to .

For integer types, this is equivalent to . For integer types, this is equivalent to .

Examples

s are treated as if having infinite size, so no filling is required and this is equivalent to .

See also , .

Unsigned right bitshift operator, . Equivalent to . See for details and examples.

– Function.

Called by syntax for constructing ranges.

Range operator. constructs a range from to with a step size of 1, and is similar but uses a step size of . These syntaxes call the function . The colon is also used in indexing to select whole dimensions.

– Function.

Construct a range by length, given a starting value and optional step (defaults to 1).

– Type.

Define an that behaves like , with the added distinction that the lower limit is guaranteed (by the type system) to be 1.

– Type.

A range where produces values of type (in the second form, is deduced automatically), parameterized by a erence value, a , and the gth. By default is the starting value , but alternatively you can supply it as the value of for some other index . In conjunction with this can be used to implement ranges that are free of roundoff error.

– Function.

Generic equality operator, giving a single result. Falls back to . Should be implemented for all types with a notion of equality, based on the abstract value that an instance represents. For example, all numeric types are compared by numeric value, ignoring type. Strings are compared as sequences of characters, ignoring encoding.

Follows IEEE semantics for floating-point numbers.

Collections should generally implement by calling recursively on all contents.

New numeric types should implement this function for two arguments of the new type, and handle comparison to other types via promotion rules where possible.

– Function.

Not-equals comparison operator. Always gives the opposite answer as . New types should generally not implement this, and rely on the fallback definition instead.

Examples

– Function.

Equivalent to .

Examples

– Function.

Less-than comparison operator. New numeric types should implement this function for two arguments of the new type. Because of the behavior of floating-point NaN values, implements a partial order. Types with a canonical partial order should implement , and types with a canonical total order should implement .

Examples

– Function.

Less-than-or-equals comparison operator.

Examples

– Function.

Greater-than comparison operator. Generally, new types should implement instead of this function, and rely on the fallback definition .

Examples

```

```

– Function.

```

```

Greater-than-or-equals comparison operator.

Examples

```

```

– Function.

```

```

Return -1, 0, or 1 depending on whether is less than, equal to, or greater than , respectively. Uses the total order implemented by . For floating-point numbers, uses but throws an error for unordered arguments.

Examples

```

```

– Function.

Bitwise not.

Examples

– Function.

Bitwise and.

Examples

– Function.

Bitwise or.

Examples

– Function.

Bitwise exclusive or of and . The infix operation is a synonym for , and can be typed by tab-completing or in the Julia REPL.

Examples

– Function.

Boolean not.

Examples

Predicate function negation: when the argument of is a function, it returns a function which computes the boolean negation of .

Examples

– Keyword.

Short-circuiting boolean AND.

– Keyword.

Short-circuiting boolean OR.

47.2 Mathematical Functions

– Function.

Inexact equality comparison: if . The default is zero and the default depends on the types of and . The keyword argument determines whether or not NaN values are considered equal (defaults to false).

For real or complex floating-point values, defaults to . This corresponds to requiring equality of about half of the significand digits. For other types, defaults to zero.

and may also be arrays of numbers, in which case defaults to but may be changed by passing a keyword argument. (For numbers, is the same thing as .) When and are arrays, if is not finite (i.e. or), the comparison falls back to checking whether all elements of and are approximately equal component-wise.

The binary operator is equivalent to with the default arguments, and is equivalent to .

Examples

– Function.

Compute sine of , where is in radians.

– Function.

Compute cosine of , where is in radians.

– Function.

Compute sine and cosine of , where is in radians.

– Function.

Compute tangent of , where is in radians.

– Function.

Compute sine of , where is in degrees.

– Function.

Compute cosine of , where is in degrees.

– Function.

Compute tangent of , where is in degrees.

– Function.

Compute $\sin(\pi x)$ more accurately than , especially for large .

– Function.

Compute $\cos(\pi x)$ more accurately than , especially for large .

– Function.

Compute hyperbolic sine of .

– Function.

Compute hyperbolic cosine of .

– Function.

Compute hyperbolic tangent of .

– Function.

Compute the inverse sine of , where the output is in radians.

– Function.

Compute the inverse cosine of , where the output is in radians

– Function.

Compute the inverse tangent of , where the output is in radians.

– Function.

Compute the inverse tangent of , using the signs of both and to determine the quadrant of the return value.

– Function.

Compute the inverse sine of , where the output is in degrees.

– Function.

Compute the inverse cosine of , where the output is in degrees.

– Function.

Compute the inverse tangent of , where the output is in degrees.

– Function.

Compute the secant of , where is in radians.

– Function.

Compute the cosecant of , where is in radians.

– Function.

Compute the cotangent of , where is in radians.

– Function.

Compute the secant of , where is in degrees.

– Function.

Compute the cosecant of , where is in degrees.

– Function.

Compute the cotangent of , where is in degrees.

– Function.

Compute the inverse secant of , where the output is in radians.

– Function.

Compute the inverse cosecant of , where the output is in radians.

– Function.

Compute the inverse cotangent of , where the output is in radians.

– Function.

Compute the inverse secant of , where the output is in degrees.

– Function.

Compute the inverse cosecant of , where the output is in degrees.

– Function.

Compute the inverse cotangent of , where the output is in degrees.

– Function.

Compute the hyperbolic secant of .

– Function.

Compute the hyperbolic cosecant of .

– Function.

Compute the hyperbolic cotangent of .

– Function.

Compute the inverse hyperbolic sine of .

– Function.

Compute the inverse hyperbolic cosine of .

– Function.

Compute the inverse hyperbolic tangent of .

– Function.

Compute the inverse hyperbolic secant of .

– Function.

Compute the inverse hyperbolic cosecant of .

– Function.

Compute the inverse hyperbolic cotangent of .

– Function.

Compute $\sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

– Function.

Compute $\cos(\pi x)/x - \sin(\pi x)/(\pi x^2)$ if $x \neq 0$, and 0 if $x = 0$. This is the derivative of .

– Function.

Convert from degrees to radians.

```

```

– Function.

```

```

Convert from radians to degrees.

```

```

– Function.

```

```

Compute the hypotenuse $\sqrt{x^2 + y^2}$ avoiding overflow and underflow.

Examples

```

```

```

```

Compute the hypotenuse $\sqrt{\sum x_i^2}$ avoiding overflow and underflow.

– Method.

```

```

Compute the natural logarithm of . Throws for negative arguments. Use complex negative arguments to obtain complex results.

There is an experimental variant in the module, which is typically faster and more accurate.

– Method.

Compute the base logarithm of . Throws for negative arguments.

Note

If is a power of 2 or 10, or should be used, as these will typically be faster and more accurate. For example,

– Function.

Compute the logarithm of to base 2. Throws for negative arguments.

Examples

– Function.

Compute the logarithm of to base 10. Throws for negative arguments.

Examples

– Function.

Accurate natural logarithm of . Throws for arguments less than -1.

There is an experimental variant in the module, which is typically faster and more accurate.

Examples

– Function.

Return such that has a magnitude in the interval $[1/2, 1)$ or 0, and is equal to $x \times 2^{exp}$.

– Function.

Compute the natural base exponential of , in other words e^x.

– Function.

Compute the base 2 exponential of , in other words 2^x.

Examples

– Function.

Compute 10^x.

Examples

– Function.

Compute $x \times 2^n$.

Examples

– Function.

Return a tuple (fpart,ipart) of the fractional and integral parts of a number. Both parts have the same sign as the argument.

Examples

– Function.

Accurately compute $e^x - 1$.

– Method.

Rounds to an integer value according to the provided , returning a value of the same type as . When not specifying a rounding mode the global mode will be used (see), which by default is round to the nearest integer (mode), with ties (fractional values of 0.5) being rounded to the nearest even integer.

Examples

The optional argument will change how the number gets rounded.

 converts the result to type , throwing an if the value is not representable.

 rounds to the specified number of digits after the decimal place (or before if negative). rounds using a base other than 10.

Examples

Note

Rounding to specified digits in bases other than 2 can be inexact when operating on binary floating point numbers. For example, the value represented by is actually *less* than 1.15, yet will be rounded to 1.2.

Chapter 48

Examples

– Type.

A type used for controlling the rounding mode of floating point operations (via / functions), or as optional arguments for rounding to the nearest integer (via the function).

Currently supported rounding modes are:

- (default)
-
-
-
- (only)
-
-

– Constant.

The default rounding mode. Rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

– Constant.

Rounds to nearest integer, with ties rounded away from zero (C/C++ behaviour).

– Constant.

Rounds to nearest integer, with ties rounded toward positive infinity (Java/JavaScript behaviour).

– Constant.

using this rounding mode is an alias for .

– Constant.

using this rounding mode is an alias for .

– Constant.

using this rounding mode is an alias for .

– Method.

Returns the nearest integral value of the same type as the complex-valued to , breaking ties using the specified s. The first is used for rounding the real components while the second is used for rounding the imaginary components.

– Function.

returns the nearest integral value of the same type as that is greater than or equal to .

converts the result to type , throwing an if the value is not representable.

and work as for .

– Function.

returns the nearest integral value of the same type as that is less than or equal to .

converts the result to type , throwing an if the value is not representable.

and work as for .

– Function.

returns the nearest integral value of the same type as whose absolute value is less than or equal to .

converts the result to type , throwing an if the value is not representable.

and work as for .

– Function.

returns the nearest integral value of type whose absolute value is less than or equal to . If the value is not representable by , an arbitrary value will be returned.

– Function.

Rounds (in the sense of) so that there are significant digits, under a base representation, default 10.

Examples

– Function.

Return the minimum of the arguments. See also the function to take the minimum element from a collection.

Examples

– Function.

Return the maximum of the arguments. See also the function to take the maximum element from a collection.

Examples

– Function.

Return . See also: that returns .

Examples

– Function.

Return if . If , return . If , return . Arguments are promoted to a common type.

– Function.

Restrict values in to the specified range, in-place. See also .

– Function.

The absolute value of .

When is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow occurs only when is applied to the minimum representable value of a signed integer. That is, when , , not as might be expected.

– Function.

Calculates , checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g.) cannot represent , thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g.) cannot represent , thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

Calculates , with the flag indicating whether overflow has occurred.

– Function.

Calculates , with the flag indicating whether overflow has occurred.

– Function.

Calculates , with the flag indicating whether overflow has occurred.

– Function.

Squared absolute value of .

– Function.

Return which has the magnitude of and the same sign as .

Examples

– Function.

Return zero if and $x/|x|$ otherwise (i.e., ±1 for real).

– Function.

Returns if the value of the sign of is negative, otherwise .

Examples

– Function.

Return with its sign flipped if is negative. For example .

– Function.

Return $\sqrt{x}$. Throws for negative arguments. Use complex negative arguments instead. The prefix operator is equivalent to .

– Function.

Integer square root: the largest integer such that .

– Function.

Return the cube root of , i.e. $x^{1/3}$. Negative values are accepted (returning the negative real root when $x < 0$).

The prefix operator is equivalent to .

– Method.

Return the real part of the complex number .

Examples

– Function.

Return the imaginary part of the complex number .

Examples

– Function.

Return both the real and imaginary parts of the complex number .

Examples

– Function.

Compute the complex conjugate of a complex number .

Examples

Returns a lazy view of the input, where each element is conjugated.

Examples

```

```

– Function.

```

```

Compute the phase angle in radians of a complex number .

Examples

```

```

– Function.

```

```

Return $\exp(iz)$.

Examples

```

```

– Function.

```

```

Number of ways to choose out of items.

Examples

```

```

– Function.

Factorial of . If is an , the factorial is computed as an integer (promoted to at least 64 bits). Note that this may overflow if is not small, but you can use to compute the result exactly in arbitrary precision. If is not an , is equivalent to .

– Function.

Greatest common (positive) divisor (or zero if and are both zero).

Examples

– Function.

Least common (non-negative) multiple.

Examples

– Function.

Computes the greatest common (positive) divisor of and and their Bézout coefficients, i.e. the integer coefficients and that satisfy $ux + vy = d = gcd(x, y)$. $gcdx(x, y)$ returns (d, u, v).

Examples

Note

Bézout coefficients are *not* uniquely defined. returns the minimal Bézout coefficients that are computed by the extended Euclidean algorithm. (Ref: D. Knuth, TAoCP, 2/e, p. 325, Algorithm X.) For signed integers, these coefficients and are minimal in the sense that $|u| < |y/d|$ and $|v| < |x/d|$. Furthermore, the signs of and are chosen so that is positive. For unsigned integers, the coefficients and might be near their , and the identity then holds only via the unsigned integers' modulo arithmetic.

– Function.

Test whether is a power of two.

Examples

– Function.

The smallest power of two not less than . Returns 0 for , and returns for negative arguments.

Examples

– Function.

The largest power of two not greater than . Returns 0 for , and returns for negative arguments.

Examples

– Function.

The smallest not less than , where is a non-negative integer. must be greater than 1, and must be greater than 0.

Examples

See also .

– Function.

The largest not greater than , where is a non-negative integer. must be greater than 1, and must not be less than 1.

Examples

See also .

– Function.

Next integer greater than or equal to that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2, etc.

Examples

– Function.

Take the inverse of modulo : such that $xy = 1 \pmod{m}$, with $div(x, y) = 0$. This is undefined for $m = 0$, or if $gcd(x, m) \neq 1$.

Examples

– Function.

Compute $x^p \pmod{m}$.

Examples

– Function.

Compute the gamma function of .

– Function.

Compute the logarithm of the absolute value of for , while for compute the principal branch cut of the logarithm of (defined for negative by analytic continuation from positive).

– Function.

Compute the logarithmic factorial of a nonnegative integer . Equivalent to of , but extends this function to non-integer .

– Function.

Euler integral of the first kind $\mathrm{B}(x, y) = \Gamma(x)\Gamma(y)/\Gamma(x + y)$.

– Function.

Natural logarithm of the absolute value of the function $\log(|\,\mathrm{B}(x, y)|)$.

– Function.

Compute the number of digits in integer written in base . The base must not be in .

Examples

```

```

– Function.

```

```

Multiply and , giving the result as a larger type.

```

```

– Macro.

```

```

Evaluate the polynomial $\sum_k c[k] z^{k-1}$ for the coefficients , , ...; that is, the coefficients are given in ascending order by power of . This macro expands to efficient inline code that uses either Horner's method or, for complex , a more efficient Goertzel-like algorithm.

```

```

– Macro.

```

```

Execute a transformed version of the expression, which calls functions that may violate strict IEEE semantics. This allows the fastest possible operation, but results are undefined – be careful when doing this, as it may change numerical results.

This sets the LLVM Fast-Math flags, and corresponds to the option in clang. See the notes on performance annotations for more details.

Examples

```

```

48.1 Statistics

– Function.

Apply the function to each element of and take the mean.

Compute the mean of whole array , or optionally along the dimensions in .

Note
Julia does not ignore values in the computation. For applications requiring the handling of missing data, the package is recommended.

– Function.

Compute the mean of over the singleton dimensions of , and write results to .

Examples

– Function.

Compute the sample standard deviation of a vector or array , optionally along dimensions in . The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of is an IID drawn from that generative distribution. This computation is equivalent to calculating . A pre-computed may be provided. If is , then the sum is scaled with , whereas the sum is scaled with if is where .

Note

Julia does not ignore values in the computation. For applications requiring the handling of missing data, the package is recommended.

– Function.

Compute the sample standard deviation of a vector with known mean . If is , then the sum is scaled with , whereas the sum is scaled with if is where .

Note

Julia does not ignore values in the computation. For applications requiring the handling of missing data, the package is recommended.

– Function.

Compute the sample variance of a vector or array , optionally along dimensions in . The algorithm will return an estimator of the generative distribution's variance under the assumption that each entry of is an IID drawn from that generative distribution. This computation is equivalent to calculating . If is , then the sum is scaled with , whereas the sum is scaled with if is where . The mean over the region may be provided.

Note

Julia does not ignore values in the computation. For applications requiring the handling of missing data, the package is recommended.

– Function.

Compute the sample variance of a collection with known mean(s) , optionally over . may contain means for each dimension of . If is , then the sum is scaled with , whereas the sum is scaled with if is where .

Note

Julia does not ignore values in the computation. For applications requiring the handling of missing data, the package is recommended.

– Function.

Compute the middle of a scalar value, which is equivalent to itself, but of the type of for consistency.

Compute the middle of two reals and , which is equivalent in both value and type to computing their mean ().

Compute the middle of a range, which consists of computing the mean of its extrema. Since a range is sorted, the mean is performed with the first and last element.

Compute the middle of an array , which consists of finding its extrema and then computing their mean.

– Function.

Compute the median of an entire array , or, optionally, along the dimensions in . For an even number of elements no exact median element exists, so the result is equivalent to calculating mean of two median elements.

Note
Julia does not ignore values in the computation. For applications requiring the handling of missing data, the package is recommended.

– Function.

Like , but may overwrite the input vector.

– Function.

Compute the quantile(s) of a vector at a specified probability or vector or tuple of probabilities . The keyword argument indicates whether can be assumed to be sorted.

The should be on the interval [0,1], and should not have any values.

Quantiles are computed via linear interpolation between the points , for where . This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R default.

Note

Julia does not ignore values in the computation. For applications requiring the handling of missing data, the package is recommended. will throw an in the presence of values in the data array.

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365

– Function.

Compute the quantile(s) of a vector at the probability or probabilities , which can be given as a single value, a vector, or a tuple. If is a vector, an optional output array may also be specified. (If not provided, a new output array is created.) The keyword argument indicates whether can be assumed to be sorted; if (the default), then the elements of may be partially sorted.

The elements of should be on the interval [0,1], and should not have any values.

Quantiles are computed via linear interpolation between the points , for where . This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R default.

Note

Julia does not ignore values in the computation. For applications requiring the handling of missing data, the package is recommended. will throw an in the presence of values in the data array.

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365

– Function.

Compute the variance of the vector . If is (the default) then the sum is scaled with , whereas the sum is scaled with if is where .

Compute the covariance matrix of the matrix along the dimension . If is (the default) then the sum is scaled with , whereas the sum is scaled with if is where .

Compute the covariance between the vectors and . If is (the default), computes $\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})^*$ where $*$ denotes the complex conjugate and . If is , computes $\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})^*$.

Compute the covariance between the vectors or matrices and along the dimension . If is (the default) then the sum is scaled with , whereas the sum is scaled with if is where .

– Function.

Return the number one.

Compute the Pearson correlation matrix of the matrix along the dimension .

Compute the Pearson correlation between the vectors and .

Compute the Pearson correlation between the vectors or matrices and along the dimension .

Chapter 49

Numbers

49.1 Standard Numeric Types

Abstract number types

– Type.

  |

  Abstract supertype for all number types.

– Type.

  |

  Abstract supertype for all real numbers.

– Type.

  |

  Abstract supertype for all floating point numbers.

– Type.

  |

  Abstract supertype for all integers.

– Type.

  |

  Abstract supertype for all signed integers.

– Type.

Abstract supertype for all unsigned integers.

Concrete number types

– Type.

16-bit floating point number type.

– Type.

32-bit floating point number type.

– Type.

64-bit floating point number type.

– Type.

Arbitrary precision floating point number type.

– Type.

Boolean type.

– Type.

8-bit signed integer type.

– Type.

8-bit unsigned integer type.

– Type.

16-bit signed integer type.

– Type.

16-bit unsigned integer type.

– Type.

32-bit signed integer type.

– Type.

32-bit unsigned integer type.

– Type.

64-bit signed integer type.

– Type.

64-bit unsigned integer type.

– Type.

128-bit signed integer type.

– Type.

128-bit unsigned integer type.

– Type.

Arbitrary precision integer type.

– Type.

Complex number type with real and imaginary part of type .

, and are aliases for , and respectively.

– Type.

Rational number type, with numerator and denominator of type .

– Type.

Irrational number type.

49.2 Data Formats

– Function.

Convert an integer to a binary string, optionally specifying a number of digits to pad to.

– Function.

Convert an integer to a hexadecimal string, optionally specifying a number of digits to pad to.

```

```

– Function.

```

```

Convert an integer to a decimal string, optionally specifying a number of digits to pad to.

Examples

```

```

– Function.

```

```

Convert an integer to an octal string, optionally specifying a number of digits to pad to.

```

```

– Function.

```

```

Convert an integer to a string in the given , optionally specifying a number of digits to pad to.

```

```

– Function.

```

```

Returns an array with element type (default) of the digits of in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indexes, such that .

Examples

– Function.

Fills an array of the digits of in the given base. More significant digits are at higher indexes. If the array length is insufficient, the least significant digits are filled up to the array length. If the array length is excessive, the excess portion is filled with zeros.

Examples

– Function.

A string giving the literal bit representation of a number.

Examples

– Method.

Parse a string as a number. If the type is an integer type, then a base can be specified (the default is 10). If the type is a floating point type, the string is parsed as a decimal floating point number. If the string does not contain a valid number, an error is raised.

– Function.

Like , but returns a of the requested type. The result will be null if the string does not contain a valid number.

– Function.

Convert a number to a maximum precision representation (typically or). See for information about some pitfalls with floating-point numbers.

– Function.

Convert a number to a signed integer. If the argument is unsigned, it is reinterpreted as signed without checking for overflow.

– Function.

Convert a number to an unsigned integer. If the argument is signed, it is reinterpreted as unsigned without checking for negative values.

Examples

– Method.

Convert a number or array to a floating point data type. When passed a string, this function is equivalent to .

– Function.

Extract the (a.k.a. mantissa), in binary representation, of a floating-point number. If is a non-zero finite number, then the result will be a number of the same type on the interval $[1, 2)$. Otherwise is returned.

Examples

– Function.

Get the exponent of a normalized floating-point number.

– Method.

Convert real numbers or arrays to complex. defaults to zero.

– Function.

Byte-swap an integer. Flip the bits of its binary representation.

Examples

– Function.

Get a hexadecimal string of the binary representation of a floating point number.

Examples

– Function.

Convert a hexadecimal string to the floating point number it represents.

– Function.

Convert an arbitrarily long hexadecimal string to its binary representation. Returns an , i.e. an array of bytes.

Examples

– Function.

Convert an array of bytes to its hexadecimal representation. All characters are in lower-case.

Examples

49.3 General Number Functions and Constants

– Function.

Return a multiplicative identity for : a value such that . Alternatively can take a type , in which case returns a multiplicative identity for any of type .

If possible, returns a value of the same type as , and returns a value of type . However, this may not be the case for types representing dimensionful quantities (e.g. time in days), since the multiplicative identity must be dimensionless. In that case, should return an identity value of the same precision (and shape, for matrices) as .

If you want a quantity that is of the same type as , or of type , even if is dimensionful, use instead.

– Function.

Returns , where is either the type of the argument or (if a type is passed) the argument. This differs from for dimensionful quantities: is dimensionless (a multiplicative identity) while is dimensionful (of the same type as , or of type).

– Function.

Get the additive identity element for the type of (can also specify the type itself).

– Constant.

The constant pi.

– Constant.

The imaginary unit.

Examples

– Constant.

The constant e.

– Constant.

Catalan's constant.

– Constant.

Euler's constant.

– Constant.

The golden ratio.

– Constant.

Positive infinity of type .

– Constant.

Positive infinity of type .

– Constant.

Positive infinity of type .

– Constant.

A not-a-number value of type .

– Constant.

A not-a-number value of type .

– Constant.

A not-a-number value of type .

– Function.

Test whether a floating point number is subnormal.

– Function.

Test whether a number is finite.

– Function.

Test whether a number is infinite.

– Function.

Test whether a floating point number is not a number (NaN).

– Function.

Return if ; if is an array, this checks whether all of the elements of are zero.

– Function.

Return if ; if is an array, this checks whether is an identity matrix.

– Function.

The result of iterative applications of to if , or applications of if .

Returns the smallest floating point number of the same type as such . If no such exists (e.g. if is or), then returns .

– Function.

Returns the largest floating point number of the same type as such . If no such exists (e.g. if is or), then returns .

– Function.

Test whether is numerically equal to some integer.

– Function.

Test whether or all its elements are numerically equal to some real number.

Examples

– Method.

Create a Float32 from . If is not exactly representable then determines how is rounded.

Examples

See for available rounding modes.

– Method.

Create a Float64 from . If is not exactly representable then determines how is rounded.

Examples

See for available rounding modes.

– Method.

Create an arbitrary precision integer. may be an (or anything that can be converted to an). The usual mathematical operators are defined for this type, and results are promoted to a .

Instances can be constructed from strings via , or using the string literal.

– Method.

Create an arbitrary precision floating point number. may be an , a or a . The usual mathematical operators are defined for this type, and results are promoted to a .

Note that because decimal literals are converted to floating point numbers when parsed, may not yield what you expect. You may instead prefer to initialize constants from strings via , or using the string literal.

– Function.

Get the current floating point rounding mode for type , controlling the rounding of basic arithmetic functions (, , , and) and type conversion.

See for available modes.

– Method.

Set the rounding mode of floating point type , controlling the rounding of basic arithmetic functions (, , , and) and type conversion. Other numerical functions may give incorrect or invalid values when using rounding modes other than the default .

Note that this may affect other types, for instance changing the rounding mode of will change the rounding mode of . See for available modes.

Warning
This feature is still experimental, and may give unexpected or incorrect values.

– Method.

Change the rounding mode of floating point type for the duration of . It is logically equivalent to:

See for available rounding modes.

Warning
This feature is still experimental, and may give unexpected or incorrect values. A known problem is the interaction with compiler optimisations, e.g.

Here the compiler is *constant folding*, that is evaluating a known constant expression at compile time, however the rounding mode is only changed at runtime, so this is not reflected in the function result. This can be avoided by moving constants outside the expression, e.g.

– Function.

Returns if operations on subnormal floating-point values (”denormals”) obey rules for IEEE arithmetic, and if they might be converted to zeros.

– Function.

If is , subsequent floating-point operations follow rules for IEEE arithmetic on subnormal values (”denormals”). Otherwise, floating-point operations are permitted (but not required) to convert subnormal inputs or outputs to zero. Returns unless but the hardware does not support zeroing of subnormal numbers.

can speed up some computations on some hardware. However, it can break identities such as .

Integers

– Function.

Number of ones in the binary representation of .

– Function.

Number of zeros in the binary representation of .

– Function.

Number of zeros leading the binary representation of .

– Function.

Number of ones leading the binary representation of .

– Function.

Number of zeros trailing the binary representation of .

– Function.

Number of ones trailing the binary representation of .

– Function.

Returns if is odd (that is, not divisible by 2), and otherwise.

– Function.

Returns is is even (that is, divisible by 2), and otherwise.

49.4 BigFloats

The type implements arbitrary-precision floating-point arithmetic using the GNU MPFR library.

– Function.

Get the precision of a floating point number, as defined by the effective number of bits in the mantissa.

– Method.

Get the precision (in bits) currently used for arithmetic.

– Function.

Set the precision (in bits) to be used for arithmetic.

Change the arithmetic precision (in bits) for the duration of . It is logically equivalent to:

Often used as

– Method.

Create a representation of as a with precision .

– Method.

Create a representation of as a with the current global precision and rounding mode .

– Method.

Create a representation of as a with precision and rounding mode .

– Method.

Create a representation of the string as a .

49.5 Random Numbers

Random number generation in Julia uses the Mersenne Twister library via objects. Julia has a global RNG, which is used by default. Other RNG types can be plugged in by inheriting the type; they can then be used to have multiple streams of random numbers. Besides , Julia also provides the RNG type, which is a wrapper over the OS provided entropy.

Most functions related to random generation accept an optional as the first argument, , which defaults to the global one if not provided. Morever, some of them accept optionally dimension specifications (which can be given as a tuple) to generate arrays of random values.

A or RNG can generate random numbers of the following types: , , , , , , , , , , , , , , , , (or complex numbers of those types). Random floating point numbers are generated uniformly in $[0, 1)$. As represents unbounded integers, the interval must be specified (e.g.).

– Function.

Reseed the random number generator. If a is provided, the RNG will give a reproducible sequence of numbers, otherwise Julia will get entropy from the system. For , the may be a non-negative integer or a vector of integers. does not support seeding.

Examples

– Type.

Create a RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers.

Examples

– Type.

Create a RNG object. Two such objects will always generate different streams of random numbers.

– Function.

Pick a random element or array of random elements from the set of values specified by ; can be

- an indexable collection (for example or),
- an or object,
- a string (considered as a collection of characters), or
- a type: the set of values to pick from is then equivalent to for integers (this is not applicable to), and to $[0, 1)$ for floating point numbers;

defaults to .

Examples

Note
The complexity of is linear in the length of , unless an optimized method with constant complexity is available, which is the case for , and . For more than a few calls, use instead, or either or as appropriate.

– Function.

Populate the array with random values. If is specified (can be a type or a collection, cf. for details), the values are picked randomly from . This is equivalent to but without allocating a new array.

Examples

```

```

– Function.

```

```

Generate a of random boolean values.

Examples

```

```

– Function.

```

```

Generate a normally-distributed random number of type with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers. The module currently provides an implementation for the types , , and (the default), and their counterparts. When the type argument is complex, the values are drawn from the circularly symmetric complex normal distribution.

Examples

```

```

– Function.

Fill the array with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the function.

Examples

– Function.

Generate a random number of type according to the exponential distribution with scale 1. Optionally generate an array of such random numbers. The module currently provides an implementation for the types , , and (the default).

Examples

– Function.

Fill the array with random numbers following the exponential distribution (with scale 1).

Examples

– Function.

Create an array of the size of initialized RNG objects. The first RNG object given as a parameter and following RNGs in the array are initialized such that a state of the RNG object in the array would be moved forward (without generating numbers) from a previous RNG object array element on a particular number of steps encoded by the jump polynomial .

Default jump polynomial moves forward RNG state by steps.

Chapter 50

Strings

– Method.

The number of characters in string .

Examples

– Method.

The number of bytes in string .

Examples

– Method.

Multiplication operator. calls this function with all arguments, i.e. .

– Method.

Repeat a string or character times. The function is an alias to this operator.

Examples

– Function.

Create a string from any values using the function.

Examples

– Method.

Repeat a string times. This can equivalently be accomplished by calling .

Examples

– Method.

Repeat a character times. This can equivalently be accomplished by calling .

Examples

– Function.

Create a string from any value using the function.

– Method.

Convert a string to a contiguous byte array representation encoded as UTF-8 bytes. This representation is often appropriate for passing strings to C.

– Function.

Convert string data between Unicode encodings. is either a or a of UTF-XX code units, where is 8, 16, or 32. indicates the encoding of the return value: to return a (UTF-8 encoded) or to return a of UTF- data. (The alias can also be used as the integer type, for converting strings used by external C libraries.)

The function succeeds as long as the input data can be reasonably represented in the target encoding; it always succeeds for conversions between UTF-XX encodings, even for invalid Unicode data.

Only conversion to/from UTF-8 is currently supported.

– Function.

Copy a string from the address of a C-style (NUL-terminated) string encoded as UTF-8. (The pointer can be safely freed afterwards.) If is specified (the length of the data in bytes), the string does not have to be NUL-terminated.

This function is labelled ”unsafe” because it will crash if is not a valid memory address to data of the requested length.

– Method.

Get the th code unit of an encoded string. For example, returns the th byte of the representation of a UTF-8 string.

– Function.

Convert a string to type and check that it contains only ASCII data, otherwise throwing an indicating the position of the first non-ASCII byte.

Examples

– Macro.

Construct a regex, such as . The regex also accepts one or more flags, listed after the ending quote, to change its behaviour:

- enables case-insensitive matching
- treats the and tokens as matching the start and end of individual lines, as opposed to the whole string.
- allows the modifier to match newlines.
- enables ”comment mode”: whitespace is enabled except when escaped with , and is treated as starting a comment.

For example, this regex has all three flags enabled:

– Macro.

Create an object from a literal string.

– Macro.

Create a object from a literal string.

– Function.

Normalize the string according to one of the four ”normal forms” of the Unicode standard: can be , , , or . Normal forms C (canonical composition) and D (canonical decomposition) convert different visually identical representations of the same abstract string into a single canonical form, with form C being more compact. Normal forms KC and KD additionally canonicalize ”compatibility equivalents”: they convert characters that are abstractly similar but visually distinct into a single canonical choice (e.g. they expand ligatures into the individual characters), with form KC being more compact.

Alternatively, finer control and additional transformations may be be obtained by calling , where any number of the following boolean keywords options (which all default to except for) are specified:

- : do not perform canonical composition
- : do canonical decomposition instead of canonical composition (is ignored if present)
- : compatibility equivalents are canonicalized
- : perform Unicode case folding, e.g. for case-insensitive string comparison
- , , or : convert various newline sequences (LF, CRLF, CR, NEL) into a linefeed (LF), line-separation (LS), or paragraph-separation (PS) character, respectively
- : strip diacritical marks (e.g. accents)
- : strip Unicode’s ”default ignorable” characters (e.g. the soft hyphen or the left-to-right marker)
- : strip control characters; horizontal tabs and form feeds are converted to spaces; newlines are also converted to spaces unless a newline-conversion flag was specified

- : throw an error if unassigned code points are found
- : enforce Unicode Versioning Stability

For example, NFKC corresponds to the options .

– Function.

Returns an iterator over substrings of that correspond to the extended graphemes in the string, as defined by Unicode UAX #29. (Roughly, these are what users would perceive as single characters, even though they may contain more than one codepoint; for example a letter combined with an accent mark is a single grapheme.)

– Method.

Returns if the given value is valid for its type, which currently can be either or .

– Method.

Returns if the given value is valid for that type. Types currently can be either or . Values for can be of type or . Values for can be of that type, or .

– Method.

Tells whether index is valid for the given string.

Examples

– Function.

Returns if the given char or integer is an assigned Unicode code point.

– Function.

Test whether a string contains a match of the given regular expression.

– Function.

Search for the first match of the regular expression in and return a object containing the match, or nothing if the match failed. The matching substring can be retrieved by accessing and the captured sequences can be retrieved by accessing The optional argument specifies an index at which to start the search.

– Function.

Search for all matches of a the regular expression in and return a iterator over the matches. If overlap is , the matching sequences are allowed to overlap indices in the original string, otherwise they must be from distinct character ranges.

– Function.

Return a vector of the matching substrings from .

– Function.

Make a string at least columns wide when printed by padding on the left with copies of .

Examples

– Function.

Make a string at least columns wide when printed by padding on the right with copies of .

Examples

```

```

– Function.

```

```

Search for the first occurrence of the given characters within the given string. The second argument may be a single character, a vector or a set of characters, a string, or a regular expression (though regular expressions are only allowed on contiguous strings, such as ASCII or UTF-8 strings). The third argument optionally specifies a starting index. The return value is a range of indexes where the matching sequence is found, such that :

 = such that , or if unmatched.

 = such that , or if unmatched.

Examples

```

```

– Function.

```

```

Similar to , but returning the last occurrence of the given characters within the given string, searching in reverse from .

Examples

```

```

– Function.

```

```

Similar to , but return only the start index at which the substring is found, or if it is not.

Examples

```

```

– Function.

Similar to , but return only the start index at which the substring is found, or if it is not.

Examples

– Method.

Determine whether the second argument is a substring of the first.

Examples

– Method.

Reverses a string.

Examples

– Function.

Search for the given pattern in , and replace each occurrence with . If is provided, replace at most occurrences. As with , the second argument may be a single character, a vector or a set of characters, a string, or a regular expression. If is a function, each occurrence is replaced with where is the matched substring. If is a regular expression and is a , then capture group references in are replaced with the corresponding matched text.

– Function.

Return an array of substrings by splitting the given string on occurrences of the given character delimiters, which may be specified in any of the formats allowed by 's second argument (i.e. a single character, collection of characters, string, or regular expression). If is omitted, it defaults to the set of all space characters, and is taken to be . The two keyword arguments are optional: they are a maximum size for the result and a flag determining whether empty fields should be kept in the result.

Examples

– Function.

Similar to , but starting from the end of the string.

Examples

– Function.

Return with any leading and trailing whitespace removed. If (a character, or vector or set of characters) is provided, instead remove characters contained in it.

Examples

```

```

– Function.

```

```

Return with any leading whitespace and delimiters removed. The default delimiters to remove are , , , , , and . If (a character, or vector or set of characters) is provided, instead remove characters contained in it.

Examples

```

```

– Function.

```

```

Return with any trailing whitespace and delimiters removed. The default delimiters to remove are , , , , , and . If (a character, or vector or set of characters) is provided, instead remove characters contained in it.

Examples

```

```

– Function.

```

```

Returns if starts with . If is a vector or set of characters, tests whether the first character of belongs to that set.

See also .

Examples

```

```

– Function.

```

```

Returns if ends with . If is a vector or set of characters, tests whether the last character of belongs to that set.

See also .

Examples

```

```

– Function.

```

```

Returns with all characters converted to uppercase.

Examples

```

```

– Function.

```

```

Returns with all characters converted to lowercase.

Examples

```

```

– Function.

```

```

Capitalizes the first character of each word in . See also to capitalize only the first character in .

Examples

```

```

– Function.

```

```

Returns with the first character converted to uppercase (technically ”title case” for Unicode). See also to capitalize the first character of every word in .

Examples

– Function.

Returns with the first character converted to lowercase.

Examples

– Function.

Join an array of into a single string, inserting the given delimiter between adjacent strings. If is given, it will be used instead of between the last two strings. For example,

Examples

can be any iterable over elements which are convertible to strings via . will be printed to .

– Function.

Remove the last character from .

Examples

– Function.

Remove a single trailing newline from a string.

Examples

– Function.

Convert a byte index to a character index with respect to string .

See also .

Examples

– Function.

Convert a character index to a byte index.

See also .

Examples

– Function.

Get the next valid string index after . Returns a value greater than at or after the end of the string.

Examples

```

```

– Function.

```

```

Get the previous valid string index before . Returns a value less than at the beginning of the string.

Examples

```

```

– Function.

```

```

Create a random string of length , consisting of characters from , which defaults to the set of upper- and lower-case letters and the digits 0-9. The optional argument specifies a random number generator, see Random Numbers.

Examples

```

```

> **Note**
> can be any collection of characters, of type or (more efficient), provided can randomly pick characters from it.

– Function.

```

```

Gives the number of columns needed to print a character.

– Function.

Gives the number of columns needed to print a string.

Examples

– Function.

Tests whether a character is alphanumeric. A character is classified as alphabetic if it belongs to the Unicode general category Letter or Number, i.e. a character whose category code begins with 'L' or 'N'.

– Function.

Tests whether a character is alphabetic. A character is classified as alphabetic if it belongs to the Unicode general category Letter, i.e. a character whose category code begins with 'L'.

– Function.

Tests whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

– Function.

Tests whether a character is a control character. Control characters are the non-printing characters of the Latin-1 subset of Unicode.

– Function.

Tests whether a character is a numeric digit (0-9).

– Function.

Tests whether a character is printable, and not a space. Any character that would cause a printer to use ink should be classified with .

– Function.

Tests whether a character is a lowercase letter. A character is classified as lowercase if it belongs to Unicode category Ll, Letter: Lowercase.

– Function.

Tests whether a character is numeric. A character is classified as numeric if it belongs to the Unicode general category Number, i.e. a character whose category code begins with 'N'.

– Function.

Tests whether a character is printable, including spaces, but not a control character.

– Function.

Tests whether a character belongs to the Unicode general category Punctuation, i.e. a character whose category code begins with 'P'.

– Function.

Tests whether a character is any whitespace character. Includes ASCII characters '\t', '\n', '\v', '\f', '\r', and ' ', Latin-1 character U+0085, and characters in Unicode category Zs.

– Function.

Tests whether a character is an uppercase letter. A character is classified as uppercase if it belongs to Unicode category Lu, Letter: Uppercase, or Lt, Letter: Titlecase.

– Function.

Tests whether a character is a valid hexadecimal digit. Note that this does not include (as in the standard prefix).

Examples

– Type.

Create a by concatenating the string representations of the arguments together.

– Function.

General escaping of traditional C and Unicode escape sequences. Any characters in are also escaped (with a back-slash). See also .

– Function.

General unescaping of traditional C and Unicode escape sequences. Reverse of .

Chapter 51

Arrays

51.1 Constructors and Types

– Type.

Supertype for -dimensional arrays (or array-like types) with elements of type . and other types are subtypes of this. See the manual section on the interface.

– Type.

Supertype for one-dimensional arrays (or array-like types) with elements of type . Alias for .

– Type.

Supertype for two-dimensional arrays (or array-like types) with elements of type . Alias for .

– Type.

-dimensional dense array with elements of type .

– Method.

Construct an uninitialized -dimensional containing elements of type . can either be supplied explicitly, as in , or be determined by the length or number of . may be a tuple or a series of integer arguments corresponding to the lengths in each dimension. If the rank is supplied explicitly, then it must match the length or number of .

Examples

```

```

– Type.

```

```

One-dimensional dense array with elements of type , often used to represent a mathematical vector. Alias for .

– Method.

```

```

Construct an uninitialized of length .

Examples

```

```

– Type.

```

```

Two-dimensional dense array with elements of type , often used to represent a mathematical matrix. Alias for .

– Method.

```

```

Construct an uninitialized of size ×.

Examples

```

```

– Method.

Construct a 1-d array of the specified type. This is usually called with the syntax . Element values can be specified using .

Examples

Retrieve the value(s) stored at the given key or index within a collection. The syntax is converted by the compiler to .

Examples

– Function.

Create an array of all zeros with the same layout as , element type and size . The argument can be skipped, which behaves like was passed. For convenience may also be passed in variadic form.

Examples

See also , .

– Function.

Create an array of all ones with the same layout as , element type and size . The argument can be skipped, which behaves like was passed. For convenience may also be passed in variadic form.

Examples

```

```

See also , .

– Type.

```

```

Space-efficient -dimensional boolean array, which stores one bit per boolean value.

– Method.

```

```

Construct an uninitialized with the given dimensions. Behaves identically to the constructor.

Examples

```

```

– Method.

```

```

Construct a generated by the given iterable object. The shape is inferred from the object.

Examples

```

```

```

```

– Function.

```

```

Create a with all values set to .

Examples

```

```

```

```

Create a with all values set to of the same shape as .

Examples

```

```

– Function.

```

```

Create a with all values set to .

Examples

```

```

Create a with all values set to of the same shape as .

Examples

– Function.

Create an array filled with the value . For example, returns a 5×5 array of floats, with each element initialized to .

Examples

If is an object reference, all elements will refer to the same object. will return an array filled with the result of evaluating once.

– Function.

Fill array with the value . If is an object reference, all elements will refer to the same object. will return filled with the result of evaluating once.

Examples

– Method.

Create an uninitialized mutable array with the given element type and size, based upon the given source array. The second and third arguments are both optional, defaulting to the given array's and . The dimensions may be specified either as a single tuple argument or as a series of integer arguments.

Custom AbstractArray subtypes may choose which specific array type is best-suited to return for the given element type and dimensionality. If they do not specialize this method, the default is an .

For example, returns an uninitialized since ranges are neither mutable nor support 2 dimensions:

Conversely, returns an uninitialized with two elements since s are both mutable and can support 1-dimensional arrays:

Since s can only store elements of type , however, if you request a different element type it will create a regular instead:

– Method.

Create an uninitialized mutable array analogous to that specified by , but with specified by the last argument. might be a type or a function.

Examples:

creates an array that ”acts like” an (and might indeed be backed by one), but which is indexed identically to . If has conventional indexing, this will be identical to , but if has unconventional indexing then the indices of the result will match .

would create a 1-dimensional logical array whose indices match those of the columns of .

would create an array of , initialized to zero, matching the indices of .

– Function.

-by- identity matrix. The default element type is .

Examples

-by- identity matrix.

-by- identity matrix. The default element type is .

Examples

Constructs an identity matrix of the same dimensions and type as .

Examples

Note the difference from .

– Function.

Construct a range of linearly spaced elements from to .

– Function.

Construct a vector of logarithmically spaced numbers from to .

– Function.

Return a vector consisting of a random subsequence of the given array , where each element of is included (in order) with independent probability . (Complexity is linear in , so this function is efficient even if is small and is large.) Technically, this process is known as ”Bernoulli sampling” of .

– Function.

Like , but the results are stored in (which is resized as needed).

51.2 Basic functions

– Function.

Returns the number of dimensions of .

Examples

– Function.

Returns a tuple containing the dimensions of . Optionally you can specify the dimension(s) you want the length of, and get the length of that dimension, or a tuple of the lengths of dimensions you asked for.

Examples

– Method.

Returns the tuple of valid indices for array .

Examples

– Method.

Returns the valid range of indices for array along dimension .

Examples

– Method.

Return the number of elements in the collection.

Use to get the last valid index of an indexable collection.

Examples

– Function.

Creates an iterable object for visiting each index of an AbstractArray in an efficient manner. For array types that have opted into fast linear indexing (like), this is simply the range . For other array types, this returns a specialized Cartesian range to efficiently index into the array with indices specified for every dimension. For other iterables, including strings and dictionaries, this returns an iterator object supporting arbitrary index types (e.g. unevenly spaced or non-integer indices).

Example for a sparse 2-d array:

If you supply more than one argument, will create an iterable object that is fast for all arguments (a if all inputs have fast linear indexing, a otherwise). If the arrays have different sizes and/or dimensionalities, returns an iterable that spans the largest range along each dimension.

– Function.

Returns a specifying the valid range of indices for where is an . For arrays with conventional indexing (indices start at 1), or any multidimensional array, this is ; however, for one-dimensional arrays with unconventional indices, this is .

Calling this function is the ”safe” way to write algorithms that exploit linear indexing.

Examples

– Type.

specifies the ”native indexing style” for array . When you define a new type, you can choose to implement either linear indexing or cartesian indexing. If you decide to implement linear indexing, then you must set this trait for your array type:

The default is .

Julia’s internal indexing machinery will automatically (and invisibly) convert all indexing operations into the preferred style using or . This allows users to access elements of your array using any indexing style, even when explicit methods have not been provided.

If you define both styles of indexing for your , this trait can be used to select the most performant indexing style. Some methods check this trait on their inputs, and dispatch to different algorithms depending on the most efficient access pattern. In particular, creates an iterator whose type depends on the setting of this trait.

– Function.

Counts the number of nonzero values in array (dense or sparse). Note that this is not a constant-time operation. For sparse matrices, one should usually use , which returns the number of stored values.

– Function.

Transform an array to its complex conjugate in-place.

See also .

Examples

– Function.

Returns the distance in memory (in number of elements) between adjacent elements in dimension .

Examples

– Function.

Returns a tuple of the memory strides in each dimension.

Examples

– Function.

Returns a tuple of subscripts into array corresponding to the linear index .

Examples

Returns a tuple of subscripts into an array with dimensions , corresponding to the linear index .

Examples

provides the indices of the maximum element.

Examples

– Function.

The inverse of , returns the linear index corresponding to the provided subscripts.

Examples

– Function.

Check that a matrix is square, then return its common dimension. For multiple arguments, return a vector.

Examples

51.3 Broadcast and vectorization

See also the dot syntax for vectorizing functions; for example, implicitly calls . Rather than relying on "vectorized" methods of functions like to operate on arrays, you should use to vectorize via .

– Function.

Broadcasts the arrays, tuples, s, nullables, and/or scalars to a container of the appropriate type and dimensions. In this context, anything that is not a subtype of , (except for s), , or is considered a scalar. The resulting container is established by the following rules:

- If all the arguments are scalars, it returns a scalar.
- If the arguments are tuples and zero or more scalars, it returns a tuple.
- If the arguments contain at least one array or , it returns an array (expanding singleton dimensions), and treats s as 0-dimensional arrays, and tuples as 1-dimensional arrays.

The following additional rule applies to arguments: If there is at least one , and all the arguments are scalars or , it returns a treating s as "containers".

A special syntax exists for broadcasting: is equivalent to , and nested calls are fused into a single broadcast loop.

Examples

– Function.

Like , but store the result of in the array. Note that is only used to store the result, and does not supply arguments to unless it is also listed in the , as in to perform .

– Macro.

Convert every function call or operator in into a ”dot call” (e.g. convert to), and convert every assignment in to a ”dot assignment” (e.g. convert to).

If you want to *avoid* adding dots for selected function calls in , splice those function calls in with . For example, is equivalent to (no dot for).

(is equivalent to a call to .)

Examples

```

```

– Function.

```

```

Broadcasts the arrays to a common size like and returns an array of the results , where goes over the positions in the broadcast result .

Examples

```

```

– Function.

Broadcasts the and arrays to a common size and stores the value from each position in at the indices in given by the same positions in .

51.4 Indexing and assignment

– Method.

Return a subset of array as specified by , where each may be an , a , or a . See the manual section on array indexing for details.

Examples

– Method.

Store values from array within some subset of as specified by .

– Method.

Copy the block of in the range of to the block of in the range of . The sizes of the two regions must match.

– Function.

Tests whether the given array has a value associated with index . Returns if the index is out of bounds, or has an undefined reference.

– Type.

Colons (:) are used to signify indexing entire objects or dimensions at once.

Very few operations are defined on Colons directly; instead they are converted by to an internal vector type () to represent the collection of indices they span before being used.

– Type.

Create a multidimensional index , which can be used for indexing a multidimensional array . In particular, is equivalent to . One can freely mix integer and indices; for example, (where and are indices and is an) can be a useful expression when writing algorithms that work along a single dimension of an array of arbitrary dimensionality.

A is sometimes produced by , and always when iterating with an explicit .

Examples

– Type.

Define a region spanning a multidimensional rectangular range of integer indices. These are most commonly encountered in the context of iteration, where will return indices equivalent to the nested loops

Consequently these can be useful for writing algorithms that work in arbitrary dimensions.

Examples

– Function.

Convert the tuple to a tuple of indices for use in indexing into array .

The returned tuple must only contain either s or s of scalar indices that are supported by array . It will error upon encountering a novel index type that it does not know how to process.

For simple index types, it defers to the unexported to process each index . While this internal function is not intended to be called directly, may be extended by custom array or index types to provide custom indexing behaviors.

More complicated index types may require more context about the dimension into which they index. To support those cases, calls , which then recursively walks through both the given tuple of indices and the dimensional indices of in tandem. As such, not all index types are guaranteed to propagate to .

– Function.

Return if the specified indices are in bounds for the given array . Subtypes of should specialize this method if they need to provide custom bounds checking behaviors; however, in many cases one can rely on 's indices and .

See also .

Examples

Throw an error if the specified indices are not in bounds for the given array .

– Function.

Return if the given is within the bounds of . Custom types that would like to behave as indices for all arrays can extend this method in order to provide a specialized bounds checking implementation.

Examples

51.5 Views (SubArrays and other view types)

– Function.

Like , but returns a view into the parent array with the given indices instead of making a copy. Calling or on the returned computes the indices to the parent array on the fly without checking bounds.

– Macro.

Creates a from an indexing expression. This can only be applied directly to a reference expression (e.g.), and should *not* be used as the target of an assignment (e.g.). See also to switch an entire block of code to use views for slicing.

– Macro.

Convert every array-slicing operation in the given expression (which may be a / block, loop, function, etc.) to return a view. Scalar indices, non-array types, and explicit calls (as opposed to) are unaffected.

Note that the macro only affects expressions that appear explicitly in the given , not array slicing that occurs in functions called by that code.

– Function.

Returns the ”parent array” of an array view type (e.g.,), or the array itself if it is not a view.

Examples

– Function.

From an array view , returns the corresponding indexes in the parent.

– Function.

Return all the data of where the index for dimension equals . Equivalent to where is in position .

Examples

– Function.

Change the type-interpretation of a block of memory. For arrays, this constructs an array with the same binary data as the given array, but with the specified element type. For example, interprets the 4 bytes corresponding to as a .

Warning

It is not allowed to an array to an element type with a larger alignment then the alignment of the array. For a normal , this is the alignment of its element type. For a reinterpreted array, this is the alignment of the it was reinterpreted from. For example, is not allowed but is allowed.

Examples

– Function.

Return an array with the same data as , but with different dimension sizes or number of dimensions. The two arrays share the same underlying data, so that setting elements of alters the values of and vice versa.

The new dimensions may be specified either as a list of arguments or as a shape tuple. At most one dimension may be specified with a , in which case its length is computed such that its product with all the specified dimensions is equal to the length of the original array . The total number of elements must not change.

– Function.

Remove the dimensions specified by from array . Elements of must be unique and within the range . must equal 1 for all in .

Examples

– Function.

Reshape the array as a one-dimensional column vector. The resulting array shares the same underlying data as , so modifying one will also modify the other.

Examples

See also .

51.6 Concatenation and permutation

– Function.

Concatenate the input arrays along the specified dimensions in the iterable . For dimensions not in , all input arrays should have the same size, which will also be the size of the output array along that dimension. For dimensions in , the size of the output array is the sum of the sizes of the input arrays along that dimension. If is a single number, the different arrays are tightly stacked along that dimension. If is an iterable containing several dimensions, this allows one to construct block diagonal matrices and their higher-dimensional analogues by simultaneously increasing several dimensions for every new input array and putting zero blocks elsewhere. For example, builds a block diagonal matrix, i.e. a block matrix with , , ... as diagonal blocks and matching zero blocks away from the diagonal.

– Function.

Concatenate along dimension 1.

Examples

– Function.

Concatenate along dimension 2.

Examples

– Function.

Horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row.

Examples

If the first argument is a single integer , then all block rows are assumed to have block columns.

– Function.

Reverse in dimension .

Examples

– Function.

Circularly shift the data in an array. The second argument is a vector giving the amount to shift in each dimension.

Examples

See also .

– Function.

Circularly shift the data in , storing the result in . specifies the amount to shift in each dimension.

The array must be distinct from the array (they cannot alias each other).

See also .

– Function.

Copy to , indexing each dimension modulo its length. and must have the same size, but can be offset in their indices; any offset results in a (circular) wraparound. If the arrays have overlapping indices, then on the domain of the overlap agrees with .

Examples

```

```

– Method.

```

```

Returns if there is at least one element in such that is .

```

```

– Method.

```

```

Return a vector of the linear indexes of the non-zeros in (determined by). A common use of this is to convert a boolean array to an array of indexes of the elements. If there are no non-zero elements of , returns an empty array.

Examples

```

```

– Method.

Return a vector of the linear indexes of where returns . If there are no such elements of , find returns an empty array.

Examples

– Function.

Return a vector of indexes for each dimension giving the locations of the non-zeros in (determined by). If there are no non-zero elements of , returns a 2-tuple of empty arrays.

Examples

– Function.

Return a tuple where and are the row and column indexes of the non-zero values in matrix , and is a vector of the non-zero values.

Examples

```

```

– Method.

```

```

Return the linear index of the first non-zero value in (determined by). Returns if no such value is found.

Examples

```

```

– Method.

```

```

Return the linear index of the first element equal to in . Returns if is not found.

Examples

```

```

– Method.

Return the linear index of the first element of for which returns . Returns if there is no such element.

Examples

– Method.

Return the linear index of the last non-zero value in (determined by). Returns if there is no non-zero value in .

Examples

– Method.

Return the linear index of the last element equal to in . Returns if there is no element of equal to .

Examples

– Method.

Return the linear index of the last element of for which returns . Returns if there is no such element.

Examples

– Method.

Find the next linear index >= of a non-zero element of , or if not found.

Examples

– Method.

Find the next linear index >= of an element of for which returns , or if not found.

Examples

– Method.

Find the next linear index >= of an element of equal to (using), or if not found.

Examples

– Method.

Find the previous linear index <= of a non-zero element of , or if not found.

Examples

– Method.

Find the previous linear index <= of an element of for which returns , or if not found.

Examples

– Method.

Find the previous linear index <= of an element of equal to (using), or if not found.

Examples

– Function.

Permute the dimensions of array . is a vector specifying a permutation of length . This is a generalization of transpose for multi-dimensional arrays. Transpose is equivalent to .

See also: .

Examples

```

```

– Function.

```

```

Permute the dimensions of array and store the result in the array . is a vector specifying a permutation of length . The preallocated array should have and is completely overwritten. No in-place permutation is supported and unexpected results will happen if and have overlapping memory regions.

See also .

– Type.

```

```

Given an AbstractArray , create a view such that the dimensions appear to be permuted. Similar to , except that no copying occurs (shares storage with).

See also: .

Examples

```

```

– Function.

```

```

Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

51.7 Array functions

– Method.

Cumulative operation along a dimension (defaults to 1). See also to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow). For common operations there are specialized variants of , see: ,

Like , but using a starting element . The first entry of the result will be .

Examples

– Function.

Cumulative operation on along a dimension, storing the result in . The dimension defaults to 1. See also .

– Function.

Cumulative product along a dimension (defaults to 1). See also to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

– Function.

Cumulative product of along a dimension, storing the result in . The dimension defaults to 1. See also .

– Function.

Cumulative sum along a dimension (defaults to 1). See also to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

– Function.

Cumulative sum of along a dimension, storing the result in . The dimension defaults to 1. See also .

– Function.

Cumulative sum along a dimension, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy. The dimension defaults to 1.

– Function.

Compute the CRC-32c checksum of the given , which can be an , a contiguous subarray thereof, or a . Optionally, you can pass a starting integer to be mixed in with the checksum. The parameter can be used to compute a checksum on data divided into chunks: performing is equivalent to the checksum of . (Technically, a little-endian checksum is computed.)

There is also a method to checksum bytes from a stream , or to checksum all the remaining bytes. Hence you can do to checksum an entire file, or to checksum an without calling .

For a , note that the result is specific to the UTF-8 encoding (a different checksum would be obtained from a different Unicode encoding). To checksum an of some other bitstype, you can do , but note that the result may be endian-dependent.

– Function.

Finite difference operator of matrix or vector . If is a matrix, compute the finite difference over a dimension (default).

Examples

– Function.

Compute differences along vector , using as the spacing between points. The default spacing is one.

Examples

– Method.

Construct an array by repeating the entries of . The i-th element of specifies the number of times that the individual entries of the i-th dimension of should be repeated. The i-th element of specifies the number of times that a slice along the i-th dimension of should be repeated. If or are omitted, no repetition is performed.

Examples

– Function.

Rotate matrix 180 degrees.

Examples

Rotate matrix 180 degrees an integer number of times. If is even, this is equivalent to a .

Examples

– Function.

Rotate matrix left 90 degrees.

Examples

Rotate matrix left 90 degrees an integer number of times. If is zero or a multiple of four, this is equivalent to a .

Examples

– Function.

Rotate matrix right 90 degrees.

Examples

Rotate matrix right 90 degrees an integer number of times. If is zero or a multiple of four, this is equivalent to a .

Examples

– Function.

Reduce 2-argument function along dimensions of . is a vector specifying the dimensions to reduce, and is the initial value to use in the reductions. For , , and the argument is optional.

The associativity of the reduction is implementation-dependent; if you need a particular associativity, e.g. left-to-right, you should write your own loop. See documentation for .

Examples

– Function.

Evaluates to the same as , but is generally faster because the intermediate array is avoided.

Examples

– Function.

Transform the given dimensions of array using function . is called on each slice of of the form . is an integer vector specifying where the colons go in this expression. The results are concatenated along the remaining dimensions. For example, if is and is 4-dimensional, is called on for all and .

Examples

```

```

– Function.

```

```

Returns the sum of all elements of , using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy.

51.8 Combinatorics

– Function.

```

```

Construct a random permutation of length . The optional argument specifies a random number generator (see Random Numbers). To randomly permute an arbitrary vector, see or .

Examples

```

```

– Function.

```

```

Construct in a random permutation of length . The optional argument specifies a random number generator (see Random Numbers). To randomly permute an arbitrary vector, see or .

Examples

```

```

– Function.

Return the inverse permutation of . If , then .

Examples

– Function.

Returns if is a valid permutation.

Examples

– Method.

Permute vector in-place, according to permutation . No checking is done to verify that is a permutation.

To return a new permutation, use . Note that this is generally faster than for large vectors.

See also .

Examples

```

```

– Function.

```

```

Like , but the inverse of the given permutation is applied.

Examples

```

```

– Function.

```

```

Construct a random cyclic permutation of length . The optional argument specifies a random number generator, see Random Numbers.

Examples

```

```

```

```

– Function.

```

```

Construct in a random cyclic permutation of length . The optional argument specifies a random number generator, see Random Numbers.

Examples

```

```

– Function.

```

```

Return a randomly permuted copy of . The optional argument specifies a random number generator (see Random Numbers). To permute in-place, see . To obtain randomly permuted indices, see .

Examples

```

```

– Function.

```

```

In-place version of : randomly permute in-place, optionally supplying the random-number generator .

Examples

```

```

– Function.

```

```

Return a copy of reversed from start to stop.

Examples

```

```

– Function.

Given an index in , return the corresponding index in so that . (This can be nontrivial in the case where is a Unicode string.)

Examples

– Function.

In-place version of .

51.9 BitArrays

s are space-efficient ”packed” boolean arrays, which store one bit per boolean value. They can be used similarly to arrays (which store one byte per boolean value), and can be converted to/from the latter via and , respectively.

– Function.

Performs a bitwise not operation on . See .

Examples

– Function.

Performs a left rotation operation on and puts the result into . controls how far to rotate the bits.

Performs a left rotation operation in-place on . controls how far to rotate the bits.

– Function.

Performs a left rotation operation, returning a new . controls how far to rotate the bits. See also .

Examples

– Function.

Performs a right rotation operation on and puts the result into . controls how far to rotate the bits.

Performs a right rotation operation in-place on . controls how far to rotate the bits.

– Function.

Performs a right rotation operation on , returning a new . controls how far to rotate the bits. See also .

Examples

51.10 Sparse Vectors and Matrices

Sparse vectors and matrices largely support the same set of operations as their dense counterparts. The following functions are specific to sparse arrays.

– Type.

Vector type for storing sparse vectors.

– Type.

Matrix type for storing sparse matrices in the Compressed Sparse Column format.

– Function.

Convert an AbstractMatrix into a sparse matrix.

Examples

Create a sparse matrix of dimensions such that . The function is used to combine duplicates. If and are not specified, they are set to and respectively. If the function is not supplied, defaults to unless the elements of are Booleans in which case defaults to . All elements of must satisfy , and all elements of must satisfy . Numerical zeros in (, ,) are retained as structural nonzeros; to drop numerical zeros, use .

For additional documentation and an expert driver, see .

Examples

```

```

– Function.

```

```

Create a sparse vector of length such that . Duplicates are combined using the function, which defaults to if no argument is provided, unless the elements of are Booleans in which case defaults to .

Examples

```

```

```

```

Create a sparse vector of length where the nonzero indices are keys from the dictionary, and the nonzero values are the values from the dictionary.

Examples

```

```

```

```

Convert a vector into a sparse vector of length .

Examples

– Function.

Returns if is sparse, and otherwise.

– Function.

Convert a sparse matrix or vector into a dense matrix or vector.

Examples

– Function.

Returns the number of stored (filled) elements in a sparse array.

Examples

– Function.

Create a sparse vector of length or sparse matrix of size . This sparse array will not contain any nonzero values. No storage will be allocated for nonzero values during construction. The type defaults to if not specified.

Examples

– Function.

Create a sparse array with the same structure as that of , but with every nonzero element having the value .

Examples

Note the difference from .

– Method.

Create a sparse identity matrix of size . When is supplied, create a sparse identity matrix of size . The type defaults to if not specified.

is equivalent to , and can be used to efficiently create a sparse multiple of the identity matrix.

– Method.

Create a sparse identity matrix with the same size as .

Examples

Note the difference from .

Create a sparse identity matrix of size . When is supplied, create a sparse identity matrix of size . The type defaults to if not specified.

is equivalent to , and can be used to efficiently create a sparse multiple of the identity matrix.

– Function.

Construct a sparse diagonal matrix. is a tuple of vectors containing the diagonals and is a tuple containing the positions of the diagonals. In the case the input contains only one diagonal, can be a vector (instead of a tuple) and can be the diagonal position (instead of a tuple), defaulting to 0 (diagonal). Optionally, and specify the size of the resulting sparse matrix.

Examples

– Function.

Create a random length sparse vector or by sparse matrix, in which the probability of any element being nonzero is independently given by (and hence the mean density of nonzeros is also exactly). Nonzero values are sampled

from the distribution specified by and have the type . The uniform distribution is used in case is not specified. The optional argument specifies a random number generator, see Random Numbers.

Examples

```

```

– Function.

```

```

Create a random sparse vector of length or sparse matrix of size by with the specified (independent) probability of any entry being nonzero, where nonzero values are sampled from the normal distribution. The optional argument specifies a random number generator, see Random Numbers.

Examples

```

```

– Function.

```

```

Return a vector of the structural nonzero values in sparse array . This includes zeros that are explicitly stored in the sparse array. The returned vector points directly to the internal nonzero storage of , and any modifications to the returned vector will mutate as well. See and .

Examples

```

```

– Function.

Return a vector of the row indices of . Any modifications to the returned vector will mutate as well. Providing access to how the row indices are stored internally can be useful in conjunction with iterating over structural nonzero values. See also and .

Examples

– Function.

Return the range of indices to the structural nonzero values of a sparse matrix column. In conjunction with and , this allows for convenient iterating over a sparse matrix :

– Method.

Removes stored numerical zeros from , optionally trimming resulting excess space from and when is .

For an out-of-place version, see . For algorithmic information, see .

– Method.

```

```

Generates a copy of and removes stored numerical zeros from that copy, optionally trimming excess space from the result's and arrays when is .

For an in-place version and algorithmic information, see .

Examples

```

```

– Method.

```

```

Removes stored numerical zeros from , optionally trimming resulting excess space from and when is .

For an out-of-place version, see . For algorithmic information, see .

– Method.

```

```

Generates a copy of and removes numerical zeros from that copy, optionally trimming excess space from the result's and arrays when is .

For an in-place version and algorithmic information, see .

Examples

```

```

– Function.

Bilaterally permute , returning (). Column-permutation 's length must match 's column count (). Row-permutation 's length must match 's row count ().

For expert drivers and additional information, see .

Examples

– Method.

Bilaterally permute , storing result () in . Stores intermediate result () in optional argument if present. Requires that none of , , and, if present, alias each other; to store result back into , use the following method lacking :

's dimensions must match those of (and), and must have enough storage to accommodate all allocated entries in (and). Column-permutation 's length must match 's column count (). Row-permutation 's length must match 's row count ().

's dimensions must match those of (and), and must have enough storage to accommodate all allocated entries in (and).

For additional (algorithmic) information, and for versions of these methods that forgo argument checking, see (unexported) parent methods and .

See also: .

Chapter 52

Tasks and Parallel Computing

52.1 Tasks

– Type.

Create a (i.e. coroutine) to execute the given function (which must be callable with no arguments). The task exits when this function returns.

Examples

In this example, is a runnable that hasn't started yet.

– Function.

Get the currently running .

– Function.

Determine whether a task has exited.

– Function.

Determine whether a task has started executing.

– Function.

Switch to the scheduler to allow another scheduled task to run. A task that calls this function is still runnable, and will be restarted immediately if there are no other runnable tasks.

A fast, unfair-scheduling version of which immediately yields to before calling the scheduler.

– Function.

Switch to the given task. The first time a task is switched to, the task's function is called with no arguments. On subsequent switches, is returned from the task's last call to . This is a low-level call that only switches tasks, not considering states or scheduling in any way. Its use is discouraged.

– Method.

Look up the value of a key in the current task's task-local storage.

– Method.

Assign a value to a key in the current task's task-local storage.

– Method.

Call the function with a modified task-local storage, in which is assigned to ; the previous value of , or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

– Type.

Create an edge-triggered event source that tasks can wait for. Tasks that call on a are suspended and queued. Tasks are woken up when is later called on the . Edge triggering means that only tasks waiting at the time is called can be woken up. For level-triggered notifications, you must keep extra state to keep track of whether a notification has happened. The type does this, and so can be used for level-triggered events.

– Function.

Wake up tasks waiting for a condition, passing them . If is (the default), all waiting tasks are woken, otherwise only one is. If is , the passed value is raised as an exception in the woken tasks.

Returns the count of tasks woken up. Returns 0 if no tasks are waiting on .

– Function.

Add a to the scheduler's queue. This causes the task to run constantly when the system is otherwise idle, unless the task performs a blocking operation such as .

If a second argument is provided, it will be passed to the task (via the return value of) when it runs again. If is , the value is raised as an exception in the woken task.

– Macro.

Wrap an expression in a and add it to the local machine's scheduler queue. Similar to except that an enclosing does NOT wait for tasks started with an .

– Macro.

Wrap an expression in a without executing it, and return the . This only creates a task, and does not run it.

– Function.

Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of .

– Type.

Constructs a with an internal buffer that can hold a maximum of objects of type . calls on a full channel block until an object is removed with .

constructs an unbuffered channel. blocks until a matching is called. And vice-versa.

Other constructors:

- : equivalent to
- : equivalent to

– Method.

|

Appends an item to the channel . Blocks if the channel is full.

For unbuffered channels, blocks until a is performed by a different task.

– Method.

|

Removes and returns a value from a . Blocks until data is available.

For unbuffered channels, blocks until a is performed by a different task.

– Method.

|

Determine whether a has a value stored to it. Returns immediately, does not block.

For unbuffered channels returns if there are tasks waiting on a .

– Method.

|

Waits for and gets the first available item from the channel. Does not remove the item. is unsupported on an unbuffered (0-size) channel.

– Method.

|

Closes a channel. An exception is thrown by:

- on a closed channel.
- and on an empty, closed channel.

– Method.

|

Associates the lifetime of with a task. Channel is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on .

The object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed Channel objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

– Function.

Uses multiple concurrent tasks to map over a collection (or multiple equal length collections). For multiple collection arguments, is applied elementwise.

 specifies the number of tasks to run concurrently. Depending on the length of the collections, if is unspecified, up to 100 tasks will be used for concurrent mapping.

 can also be specified as a zero-arg function. In this case, the number of tasks to run in parallel is checked before processing every element and a new task started if the value of is less than the current number of tasks.

If is specified, the collection is processed in batch mode. must then be a function that must accept a of argument tuples and must return a vector of results. The input vector will have a length of or less.

The following examples highlight execution in different tasks by returning the of the tasks in which the mapping function is executed.

First, with undefined, each element is processed in a different task.

With all elements are processed in 2 tasks.

With defined, the mapping function needs to be changed to accept an array of argument tuples and return an array of results. is used in the modified mapping function to achieve this.

Note

Currently, all tasks in Julia are executed in a single OS thread co-operatively. Consequently, is beneficial only when the mapping function involves any I/O - disk, network, remote worker invocation, etc.

– Function.

Like , but stores output in rather than returning a collection.

52.2 General Parallel Computing Support

– Function.

Launches worker processes via the specified cluster manager.

For example, Beowulf clusters are supported via a custom cluster manager implemented in the package .

The number of seconds a newly launched worker waits for connection establishment from the master can be specified via variable in the worker process's environment. Relevant only when using TCP/IP as transport.

Add processes on remote machines via SSH. Requires to be installed in the same location on each node, or to be available via a shared file system.

is a vector of machine specifications. Workers are started for each specification.

A machine specification is either a string or a tuple - .

is a string of the form . defaults to current user, to the standard ssh port. If is specified, other workers will connect to this worker at the specified and .

is the number of workers to be launched on the specified host. If specified as it will launch as many workers as the number of cores on the specific host.

Keyword arguments:

- : if then SSH tunneling will be used to connect to the worker from the master process. Default is .
- : specifies additional ssh options, e.g. ‘
- : specifies the maximum number of workers connected to in parallel at a host. Defaults to 10.
- : specifies the working directory on the workers. Defaults to the host's current directory (as found by)
- : if then BLAS will run on multiple threads in added processes. Default is .
- : name of the executable. Defaults to or as the case may be.
- : additional flags passed to the worker processes.
- : Specifies how the workers connect to each other. Sending a message between unconnected workers results in an error.
  - : All processes are connected to each other. The default.
  - : Only the driver process, i.e. 1 connects to the workers. The workers do not connect to each other.
  - : The method of the cluster manager specifies the connection topology via fields and in . A worker with a cluster manager identity will connect to all workers specified in .
- : Applicable only with . If , worker-worker connections are setup lazily, i.e. they are setup at the first instance of a remote call between workers. Default is true.

Environment variables :

If the master process fails to establish a connection with a newly launched worker within 60.0 seconds, the worker treats it as a fatal situation and terminates. This timeout can be controlled via environment variable . The value of on the master process specifies the number of seconds a newly launched worker waits for connection establishment.

Equivalent to

Note that workers do not run a startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

|

Launches workers using the in-built which only launches workers on the local host. This can be used to take advantage of multiple cores. will add 4 processes on the local machine. If is , binding is restricted to . Keyword args , , , , and have the same effect as documented for .

– Function.

|

Get the number of available processes.

– Function.

|

Get the number of available worker processes. This is one less than . Equal to if .

– Method.

|

Returns a list of all process identifiers.

– Method.

|

Returns a list of all process identifiers on the same physical node. Specifically all workers bound to the same ip-address as are returned.

– Function.

|

Returns a list of all worker process identifiers.

– Function.

|

Removes the specified workers. Note that only process 1 can add or remove workers.

Argument specifies how long to wait for the workers to shut down: - If unspecified, will wait until all requested are removed. - An is raised if all workers cannot be terminated before the requested seconds. - With a value of 0, the call returns immediately with the workers scheduled for removal in a different task. The scheduled object is returned. The user should call on the task before invoking any other parallel calls.

– Function.

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

– Function.

Get the id of the current process.

– Function.

Transform collection by applying to each element using available workers and tasks.

For multiple collection arguments, apply elementwise.

Note that must be made available to all worker processes; see Code Availability and Loading Packages for details.

If a worker pool is not specified, all available workers, i.e., the default worker pool is used.

By default, distributes the computation over all specified workers. To use only the local process and distribute over tasks, specify . This is equivalent to using . For example, is equivalent to

can also use a mix of processes and tasks via the argument. For batch sizes greater than 1, the collection is processed in multiple batches, each of length or less. A batch is sent as a single request to a free worker, where a local processes elements from the batch using multiple concurrent tasks.

Any error stops from processing the remainder of the collection. To override this behavior you can specify an error handling function via argument which takes in a single argument, i.e., the exception. The function can stop the processing by rethrowing the error, or, to continue, return any value which is then returned inline with the results to the caller.

Consider the following two examples. The first one returns the exception object inline, the second a 0 in place of any exception:

```

```

Errors can also be handled by retrying failed computations. Keyword arguments and are passed through to as keyword arguments and respectively. If batching is specified, and an entire batch fails, all items in the batch are retried.

Note that if both and are specified, the hook is called before retrying. If does not throw (or rethrow) an exception, the element will not be retried.

Example: On errors, retry on an element a maximum of 3 times without any delay between retries.

```

```

Example: Retry only if the exception is not of type , with exponentially increasing delays up to 3 times. Return a in place for all occurrences.

```

```

– Type.

```

```

Exceptions on remote computations are captured and rethrown locally. A wraps the of the worker and a captured exception. A captures the remote exception and a serializable form of the call stack when the exception was raised.

– Type.

```

```

Create a on process . The default is the current process.

– Method.

```

```

Make a reference to a on process . The default is the current process.

– Method.

|

Create references to remote channels of a specific size and type. is a function that when executed on must return an implementation of an .

For example, , will return a reference to a channel of type and size 10 on .

The default is the current process.

– Function.

|

Block the current task until some event occurs, depending on the type of the argument:

- : Wait for a value to become available on the specified remote channel.
- : Wait for a value to become available for the specified future.
- : Wait for a value to be appended to the channel.
- : Wait for on a condition.
- : Wait for a process or process chain to exit. The field of a process can be used to determine success or failure.
- : Wait for a to finish, returning its result value. If the task fails with an exception, the exception is propagated (re-thrown in the task that called).
- : Wait for changes on a file descriptor (see for keyword arguments and return code)

If no argument is passed, the task blocks for an undefined period. A task can only be restarted by an explicit call to or .

Often is called within a loop to ensure a waited-for condition is met before proceeding.

– Method.

|

Waits and fetches a value from depending on the type of :

- : Wait for and get the value of a . The fetched value is cached locally. Further calls to on the same reference return the cached value. If the remote value is an exception, throws a which captures the remote exception and backtrace.
- : Wait for and get the value of a remote reference. Exceptions raised are same as for a .

Does not remove the item fetched.

– Method.

|

Call a function asynchronously on the given arguments on the specified process. Returns a . Keyword arguments, if any, are passed through to .

– Method.

Perform a faster in one message on the specified by worker id . Keyword arguments, if any, are passed through to .

See also and .

– Method.

Perform in one message. Keyword arguments, if any, are passed through to . Any remote exceptions are captured in a and thrown.

See also and .

– Method.

Executes on worker asynchronously. Unlike , it does not store the result of computation, nor is there a way to wait for its completion.

A successful invocation indicates that the request has been accepted for execution on the remote node.

While consecutive s to the same worker are serialized in the order they are invoked, the order of executions on the remote worker is undetermined. For example, will serialize the call to , followed by and in that order. However, it is not guaranteed that is executed before on worker 2.

Any exceptions thrown by are printed to on the remote worker.

Keyword arguments, if any, are passed through to .

– Method.

Store a set of values to the . If the channel is full, blocks until space is available. Returns its first argument.

– Method.

Store a value to a . s are write-once remote references. A on an already set throws an . All asynchronous remote calls return s and set the value to the return value of the call upon completion.

– Method.

Fetch value(s) from a , removing the value(s) in the processs.

– Method.

Determine whether a has a value stored to it. Note that this function can cause race conditions, since by the time you receive its result it may no longer be true. However, it can be safely used on a since they are assigned only once.

– Method.

Determine whether a has a value stored to it.

If the argument is owned by a different node, this call will block to wait for the answer. It is recommended to wait for in a separate task instead or to use a local as a proxy:

– Type.

Create a WorkerPool from a vector of worker ids.

– Type.

An implementation of an . , , (and other remote calls which execute functions remotely) benefit from caching the serialized/deserialized functions on the worker nodes, especially closures (which may capture large amounts of data).

The remote cache is maintained for the lifetime of the returned object. To clear the cache earlier, use .

For global variables, only the bindings are captured in a closure, not the data. blocks can be used to capture global data.

Examples

The above would transfer only once to each worker.

– Function.

containing idle - used by and (by default).

– Method.

Removes all cached functions from all participating workers.

– Function.

Returns an anonymous function that executes function on an available worker using .

– Method.

variant of . Waits for and takes a free worker from and performs a on it.

– Method.

variant of . Waits for and takes a free worker from and performs a on it.

– Method.

variant of . Waits for and takes a free worker from and performs a on it.

– Method.

variant of . Waits for and takes a free worker from and performs a on it.

– Function.

Waits until returns or for seconds, whichever is earlier. is polled every seconds.

– Macro.

Create a closure around an expression and run it on an automatically-chosen process, returning a to the result.

Examples

– Macro.

Create a closure around an expression and run the closure asynchronously on process . Returns a to the result. Accepts two arguments, and an expression.

Examples

– Macro.

Equivalent to . See and .

Examples

– Macro.

Equivalent to . See and .

Examples

– Macro.

Like , wraps an expression in a and adds it to the local machine's scheduler queue. Additionally it adds the task to the set of items that the nearest enclosing waits for.

– Macro.

Wait until all dynamically-enclosed uses of , , and are complete. All exceptions thrown by enclosed async operations are collected and thrown as a .

– Macro.

A parallel for loop of the form :

The specified range is partitioned and locally executed across all workers. In case an optional reducer function is specified, performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, executes asynchronously, i.e. it spawns independent tasks on all available workers and returns immediately without waiting for completion. To wait for completion, prefix the call with , like :

– Macro.

Execute an expression under on all . Errors on any of the processes are collected into a and thrown. For example:

will define on all processes.

Unlike and , does not capture any local variables. Instead, local variables can be broadcast using interpolation:

The optional argument allows specifying a subset of all processes to have execute the expression.

Equivalent to calling .

– Method.

Clears global bindings in modules by initializing them to . should be of type or a collection of s . and identify the processes and the module in which global variables are to be reinitialized. Only those names found to be defined under are cleared.

An exception is raised if a global constant is requested to be cleared.

– Function.

s and s are identified by fields:

- - refers to the node where the underlying object/storage referred to by the reference actually exists.
- - refers to the node the remote reference was created from. Note that this is different from the node where the underlying object referred to actually exists. For example calling from the master process would result in a value of 2 and a value of 1.
- is unique across all references created from the worker specified by .

Taken together, and uniquely identify a reference across all workers.

is a low-level API which returns a object that wraps and values of a remote reference.

– Function.

A low-level API which returns the backing for an returned by . The call is valid only on the node where the backing channel exists.

– Function.

A low-level API which given a connection or a , returns the of the worker it is connected to. This is useful when writing custom methods for a type, which optimizes the data written out depending on the receiving process id.

– Method.

Returns the cluster cookie.

– Method.

Sets the passed cookie as the cluster cookie, then returns it.

52.3 Shared Arrays

– Type.

Construct a of a bits type and size across the processes specified by - all of which have to be on the same host. If is specified by calling , then must match the length of .

If is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, and will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

If an function of the type is specified, it is called on all the participating workers.

The shared array is valid as long as a reference to the object exists on the node which created the mapping.

Construct a backed by the file , with element type (must be a bits type) and size , across the processes specified by - all of which have to be on the same host. This file is mmapped into the host memory, with the following consequences:

- The array data must be represented in binary format (e.g., an ASCII format like CSV cannot be supported)
- Any changes you make to the array values (e.g.,) will also change the values on disk

If is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, and will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

must be one of , , , or , and defaults to if the file specified by already exists, or if not. If an function of the type is specified, it is called on all the participating workers. You cannot specify an function if the file is not writable.

allows you to skip the specified number of bytes at the beginning of the file.

– Method.

Get the vector of processes mapping the shared array.

– Function.

Returns the actual object backing .

– Function.

Returns the current worker's index in the list of workers mapping the (i.e. in the same list returned by), or 0 if the is not mapped locally.

– Function.

Returns a range describing the "default" indexes to be handled by the current process. This range should be interpreted in the sense of linear indexing, i.e., as a sub-range of . In multi-process contexts, returns an empty range in the parent process (or any process for which returns 0).

It's worth emphasizing that exists purely as a convenience, and you can partition work on the array among workers any way you wish. For a , all indexes should be equally fast for each worker process.

52.4 Multi-Threading

This experimental interface supports Julia's multi-threading capabilities. Types and functions described here might (and likely will) change in the future.

– Function.

Get the ID number of the current thread of execution. The master thread has ID .

– Function.

Get the number of threads available to the Julia process. This is the inclusive upper bound on .

– Macro.

A macro to parallelize a for-loop to run with multiple threads. This spawns number of threads, splits the iteration space amongst them, and iterates in parallel. A barrier is placed at the end of the loop which waits for all the threads to finish execution, and the loop returns.

– Type.

Holds a reference to an object of type , ensuring that it is only accessed atomically, i.e. in a thread-safe manner.

Only certain "simple" types can be used atomically, namely the primitive integer and float-point types. These are ..., ..., and ....

New atomic objects can be created from a non-atomic values; if none is specified, the atomic object is initialized with zero.

Atomic objects can be accessed using the notation:

Examples

Atomic operations use an prefix, such as , , etc.

– Function.

Atomically compare-and-set

Atomically compares the value in with . If equal, write to . Otherwise, leaves unmodified. Returns the old value in . By comparing the returned value to (via) one knows whether was modified and now holds the new value .

For further details, see LLVM's instruction.

This function can be used to implement transactional semantics. Before the transaction, one records the value in . After the transaction, the new value is stored only if has not been modified in the mean time.

Examples

– Function.

Atomically exchange the value in

Atomically exchanges the value in with . Returns the **old** value.

For further details, see LLVM's instruction.

Examples

– Function.

Atomically add to

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

Examples

– Function.

Atomically subtract from

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

Examples

– Function.

Atomically bitwise-and with

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

Examples

– Function.

Atomically bitwise-nand (not-and) with

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

Examples

– Function.

Atomically bitwise-or with

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

Examples

– Function.

Atomically bitwise-xor (exclusive-or) with

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

Examples

– Function.

Atomically store the maximum of and in

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

Examples

– Function.

Atomically store the minimum of and in

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

Examples

– Function.

Insert a sequential-consistency memory fence

Inserts a memory fence with sequentially-consistent ordering semantics. There are algorithms where this is needed, i.e. where an acquire/release ordering is insufficient.

This is likely a very expensive operation. Given that all other atomic operations in Julia already have acquire/release semantics, explicit fences should not be necessary in most cases.

For further details, see LLVM's instruction.

52.5 ccall using a threadpool (Experimental)

– Macro.

The macro is called in the same way as but does the work in a different thread. This is useful when you want to call a blocking C function without causing the main thread to become blocked. Concurrency is limited by size of the libuv thread pool, which defaults to 4 threads but can be increased by setting the environment variable and restarting the process.

Note that the called function should never call back into Julia.

52.6 Synchronization Primitives

– Type.

Abstract supertype describing types that implement the thread-safe synchronization primitives: , , , and

– Function.

Acquires the lock when it becomes available. If the lock is already locked by a different task/thread, it waits for it to become available.

Each must be matched by an .

– Function.

Releases ownership of the lock.

If this is a recursive lock which has been acquired before, it just decrements an internal counter and returns immediately.

– Function.

Acquires the lock if it is available, returning if successful. If the lock is already locked by a different task/thread, returns .

Each successful must be matched by an .

– Function.

Check whether the lock is held by any task/thread. This should not be used for synchronization (see instead).

– Type.

Creates a reentrant lock for synchronizing Tasks. The same task can acquire the lock as many times as required. Each must be matched with an .

This lock is NOT threadsafe. See for a threadsafe lock.

– Type.

These are standard system mutexes for locking critical sections of logic.

On Windows, this is a critical section object, on pthreads, this is a .

See also SpinLock for a lighter-weight lock.

– Type.

Creates a non-reentrant lock. Recursive use will result in a deadlock. Each must be matched with an .

Test-and-test-and-set spin locks are quickest up to about 30ish contending threads. If you have more contention than that, perhaps a lock is the wrong way to synchronize.

See also RecursiveSpinLock for a version that permits recursion.

See also Mutex for a more efficient version on one core or if the lock may be held for a considerable length of time.

– Type.

Creates a reentrant lock. The same thread can acquire the lock as many times as required. Each must be matched with an .

See also SpinLock for a slightly faster version.

See also Mutex for a more efficient version on one core or if the lock may be held for a considerable length of time.

– Type.

Creates a counting semaphore that allows at most acquires to be in use at any time. Each acquire must be mached with a release.

This construct is NOT threadsafe.

– Function.

Wait for one of the permits to be available, blocking until one can be acquired.

– Function.

Return one permit to the pool, possibly allowing another task to acquire it and resume execution.

52.7 Cluster Manager Interface

This interface provides a mechanism to launch and manage Julia workers on different cluster environments. There are two types of managers present in Base: , for launching additional workers on the same host, and , for launching on remote hosts via . TCP/IP sockets are used to connect and transport messages between processes. It is possible for Cluster Managers to provide a different transport.

– Function.

Implemented by cluster managers. For every Julia worker launched by this function, it should append a entry to and notify . The function MUST exit once all workers, requested by have been launched. is a dictionary of all keyword arguments was called with.

– Function.

Implemented by cluster managers. It is called on the master process, during a worker's lifetime, with appropriate values:

- with / when a worker is added / removed from the Julia worker pool.
- with when is called. The should signal the appropriate worker with an interrupt signal.
- with for cleanup purposes.

– Method.

Implemented by cluster managers. It is called on the master process, by . It should cause the remote worker specified by to exit. executes a remote on .

– Method.

Implemented by cluster managers using custom transports. It should establish a logical connection to worker with id , specified by and return a pair of objects. Messages from to current process will be read off , while messages to be sent to will be written to . The custom transport implementation must ensure that messages are delivered and received completely and in order. sets up TCP/IP socket connections in-between workers.

– Function.

Called by cluster managers implementing custom transports. It initializes a newly launched process as a worker. Command line argument has the effect of initializing a process as a worker using TCP/IP sockets for transport. is a .

– Function.

is an internal function which is the default entry point for worker processes connecting via TCP/IP. It sets up the process as a Julia cluster worker.

If the cookie is unspecified, the worker tries to read it from its STDIN.

host:port information is written to stream (defaults to STDOUT).

The function closes STDIN (after reading the cookie if required), redirects STDERR to STDOUT, listens on a free port (or if specified, the port in the command line option) and schedules tasks to process incoming TCP connections and requests.

It does not return.

– Function.

Called by cluster managers using custom transports. It should be called when the custom transport implementation receives the first message from a remote worker. The custom transport must manage a logical connection to the remote worker and provide two objects, one for incoming messages and the other for messages addressed to the remote worker. If is , the remote peer initiated the connection. Whichever of the pair initiates the connection sends the cluster cookie and its Julia version number to perform the authentication handshake.

See also .

Chapter 53

Linear Algebra

53.1 Standard Functions

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse factorizations call functions from SuiteSparse.

– Method.

Multiplication operator. calls this function with all arguments, i.e. .

– Method.

Left division operator: multiplication of by the inverse of on the left. Gives floating-point results for integer arguments.

Examples

– Function.

Compute the dot product between two vectors. For complex vectors, the first vector is conjugated. When the vectors have equal lengths, calling is semantically equivalent to .

Examples

– Function.

For any iterable containers and (including arrays of any dimension) of numbers (or any element type for which is defined), compute the Euclidean dot product (the sum of) as if they were vectors.

Examples

– Function.

Compute the cross product of two 3-vectors.

Examples

```

```

– Function.

```

```

Compute a convenient factorization of , based upon the type of the input matrix. checks to see if it is symmetric/-triangular/etc. if is passed as a generic matrix. checks every element of to verify/rule out each property. It will short-circuit as soon as it can rule out symmetry/triangular structure. The return value can be reused for efficient solving of multiple systems. For example: .

Properties of	type of factorization
Positive-definite	Cholesky (see)
Dense Symmetric/Hermitian	Bunch-Kaufman (see)
Sparse Symmetric/Hermitian	LDLt (see)
Triangular	Triangular
Diagonal	Diagonal
Bidiagonal	Bidiagonal
Tridiagonal	LU (see)
Symmetric real tridiagonal	LDLt (see)
General square	LU (see)
General non-square	QR (see)

If is called on a Hermitian positive-definite matrix, for instance, then will return a Cholesky factorization.

Examples

```

```

This returns a , which can now be passed to other linear algebra functions (e.g. eigensolvers) which will use specialized methods for types.

– Type.

Construct a matrix from the diagonal of .

Examples

Construct a matrix with as its diagonal.

Examples

– Type.

Constructs an upper () or lower () bidiagonal matrix using the given diagonal () and off-diagonal () vectors. The result is of type and provides efficient specialized linear solvers, but may be converted into a regular matrix with (or for short). The length of must be one less than the length of .

Examples

Construct a matrix from the main diagonal of and its first super- (if) or sub-diagonal (if).

Examples

– Type.

Construct a symmetric tridiagonal matrix from the diagonal () and first sub/super-diagonal (), respectively. The result is of type and provides efficient specialized eigensolvers, but may be converted into a regular matrix with (or for short).

Examples

Construct a symmetric tridiagonal matrix from the diagonal and first sub/super-diagonal, of the symmetric matrix .

Examples

– Type.

Construct a tridiagonal matrix from the first subdiagonal, diagonal, and first superdiagonal, respectively. The result is of type and provides efficient specialized linear solvers, but may be converted into a regular matrix with (or for short). The lengths of and must be one less than the length of .

Examples

returns a array based on (abstract) matrix , using its first lower diagonal, main diagonal, and first upper diagonal.

Examples

– Type.

Construct a view of the upper (if) or lower (if) triangle of the matrix .

Examples

Note that will not be equal to unless is itself symmetric (e.g. if).

– Type.

Construct a view of the upper (if) or lower (if) triangle of the matrix .

Examples

Note that will not be equal to unless is itself Hermitian (e.g. if).

– Type.

Construct a view of the the matrix .

Examples

– Type.

Construct an view of the the matrix .

Examples

– Type.

Generically sized uniform scaling operator defined as a scalar times the identity operator, . See also .

Examples

– Function.

Compute the LU factorization of , such that . By default, pivoting is used. This can be overridden by passing for the second argument.

See also .

Examples

– Function.

Compute the LU factorization of a sparse matrix .

For sparse with real or complex element type, the return type of is , with = or respectively and is an integer type (or).

The individual components of the factorization can be accessed by indexing:

Component	Description
	(lower triangular) part of
	(upper triangular) part of
	right permutation
	left permutation
	of scaling factors
	components

The relation between and is

further supports the following functions:

-
-
-

Note

uses the UMFPACK library that is part of SuiteSparse. As this library only supports sparse matrices with or elements, converts into a copy that is of type or as appropriate.

Compute the LU factorization of .

In most cases, if is a subtype of with an element type supporting , , and , the return type is . If pivoting is chosen (default) the element type should also support and .

The individual components of the factorization can be accessed by indexing:

Component	Description
	(lower triangular) part of
	(upper triangular) part of
	(right) permutation
	(right) permutation

The relationship between and is

further supports the following functions:

Supported function		

Examples

– Function.

```
```

is the same as , but saves space by overwriting the input , instead of creating a copy. An exception is thrown if the factorization produces a number not representable by the element type of , e.g. for integer types.

– Function.

```
```

Compute the Cholesky factorization of a positive definite matrix and return the matrix such that .

Examples

```
```

```
```

Compute the square root of a non-negative number .

Examples

```
```

```
```

Compute the square root of a non-negative UniformScaling .

Examples

```
```

– Function.

Compute the Cholesky factorization of a sparse positive definite matrix . must be a or a / view of a . Note that even if doesn't have the type tag, it must still be symmetric or Hermitian. A fill-reducing permutation is used. is most frequently used to solve systems of equations with , but also the methods , , and are defined for . You can also extract individual factors from , using . However, since pivoting is on by default, the factorization is internally represented as with a permutation matrix ; using just without accounting for will give incorrect answers. To include the effects of permutation, it's typically preferable to extract "combined" factors like (the equivalent of) and (the equivalent of).

Setting the optional keyword argument computes the factorization of instead of . If the argument is nonempty, it should be a permutation of giving the ordering to use (instead of CHOLMOD's default AMD ordering).

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to or as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the module.

Compute the Cholesky factorization of a dense symmetric positive definite matrix and return a factorization. The matrix can either be a or or a *perfectly* symmetric or Hermitian . The triangular Cholesky factor can be obtained from the factorization with: and . The following functions are available for objects: , , , , and .

Examples

Compute the pivoted Cholesky factorization of a dense symmetric positive semi-definite matrix and return a factorization. The matrix can either be a or or a *perfectly* symmetric or Hermitian . The triangular Cholesky factor can be obtained from the factorization with: and . The following functions are available for objects: , , , and . The argument determines the tolerance for determining the rank. For negative values, the tolerance is the machine precision.

– Function.

Compute the Cholesky (LL') factorization of , reusing the symbolic factorization . must be a or a / view of a . Note that even if doesn't have the type tag, it must still be symmetric or Hermitian.

See also .

> **Note**
> This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to or as appropriate.

The same as , but saves space by overwriting the input , instead of creating a copy. An exception is thrown if the factorization produces a number not representable by the element type of , e.g. for integer types.

Examples

The same as , but saves space by overwriting the input , instead of creating a copy. An exception is thrown if the factorization produces a number not representable by the element type of , e.g. for integer types.

– Function.

Update a Cholesky factorization with the vector . If then but the computation of only uses operations.

– Function.

Downdate a Cholesky factorization with the vector . If then but the computation of only uses operations.

– Function.

Update a Cholesky factorization with the vector . If then but the computation of only uses operations. The input factorization is updated in place such that on exit . The vector is destroyed during the computation.

– Function.

Downdate a Cholesky factorization with the vector . If then but the computation of only uses operations. The input factorization is updated in place such that on exit . The vector is destroyed during the computation.

– Function.

Compute the LDL' factorization of a sparse matrix . must be a or a / view of a . Note that even if doesn't have the type tag, it must still be symmetric or Hermitian. A fill-reducing permutation is used. is most frequently used to solve systems of equations with . The returned factorization object also supports the methods , , , and . You can extract individual factors from using . However, since pivoting is on by default, the factorization is internally represented as with a permutation matrix ; using just without accounting for will give incorrect answers. To include the effects of permutation, it is typically preferable to extract "combined" factors like (the equivalent of) and (the equivalent of). The complete list of supported factors is .

Setting the optional keyword argument computes the factorization of instead of . If the argument is nonempty, it should be a permutation of giving the ordering to use (instead of CHOLMOD's default AMD ordering).

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to or as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the module.

Compute an factorization of a real symmetric tridiagonal matrix such that where is a unit lower triangular matrix and is a vector. The main use of an factorization is to solve the linear system of equations with .

– Function.

Compute the LDL' factorization of , reusing the symbolic factorization . must be a or a / view of a . Note that even if doesn't have the type tag, it must still be symmetric or Hermitian.

See also .

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to or as appropriate.

Same as , but saves space by overwriting the input , instead of creating a copy.

– Function.

Compute the (pivoted) QR factorization of such that either or . Also see . The default is to compute a thin factorization. Note that is not extended with zeros when the full is requested.

Computes the polar decomposition of a vector. Returns , a unit vector in the direction of , and , the norm of .

See also , , and .

Examples

– Function.

Computes the polar decomposition of a vector. Instead of returning a new vector as , this function mutates the input vector in place. Returns , a unit vector in the direction of (this is a mutation of), and , the norm of .

See also , , and .

– Function.

Compute the QR factorization of the matrix : an orthogonal (or unitary if is complex-valued) matrix , and an upper triangular matrix such that

$$A = QR$$

The returned object stores the factorization in a packed format:

- if then is a object,
- otherwise if the element type of is a BLAS type (, , or), then is a object,
- otherwise is a object.

The individual components of the factorization can be accessed by indexing with a symbol:

- : the orthogonal/unitary matrix
- : the upper triangular matrix
- : the permutation vector of the pivot (only)
- : the permutation matrix of the pivot (only)

The following functions are available for the objects: , , and . When is rectangular, will return a least squares solution and if the solution is not unique, the one with smallest norm is returned.

Multiplication with respect to either thin or full is allowed, i.e. both and are supported. A matrix can be converted into a regular matrix with which has a named argument .

Examples

Note

returns multiple types because LAPACK uses several representations that minimize the memory storage requirements of products of Householder elementary reflectors, so that the and matrices can be stored compactly rather as two separate dense matrices.

Compute the factorization of a sparse matrix . Fill-reducing row and column permutations are used such that . The main application of this type is to solve least squares or underdetermined problems with . The function calls the C library SPQR.

Examples

```

```

– Function.

```

```

is the same as when is a subtype of , but saves space by overwriting the input , instead of creating a copy. An exception is thrown if the factorization produces a number not representable by the element type of , e.g. for integer types.

– Type.

```

```

A QR matrix factorization stored in a packed format, typically obtained from . If A is an × matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors v_i and coefficients τ_i where:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

The object has two fields:

- is an × matrix.
  - The upper triangular part contains the elements of R, that is for a object .
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix .
- is a vector of length containing the coefficients au_i.

– Type.

A QR matrix factorization stored in a compact blocked format, typically obtained from . If A is an × matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. It is similar to the format except that the orthogonal/unitary matrix Q is stored in *Compact WY* format [1], as a lower trapezoidal matrix V and an upper triangular matrix T where

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T) = I - VTV^T$$

such that v_i is the ith column of V, and au_i is the ith diagonal element of T.

The object has two fields:

- , as in the type, is an × matrix.
  - The upper triangular part contains the elements of R, that is for a object .
  - The subdiagonal part contains the reflectors v_i stored in a packed format such that .
- is a square matrix with columns, whose upper triangular part gives the matrix T above (the subdiagonal elements are ignored).

  Note
  This format should not to be confused with the older *WY* representation [2].

[2] C Bischof and C Van Loan, "The WY representation for products of Householder matrices", SIAM J Sci Stat Comput 8 (1987), s2-s13. doi:10.1137/0908009

[1] R Schreiber and C Van Loan, "A storage-efficient WY representation for products of Householder transformations", SIAM J Sci Stat Comput 10 (1989), 53-57. doi:10.1137/0910005

– Type.

A QR matrix factorization with column pivoting in a packed format, typically obtained from . If A is an × matrix, then

$$AP = QR$$

where P is a permutation matrix, Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

The object has three fields:

- is an × matrix.
  - The upper triangular part contains the elements of R, that is for a object .
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix .
- is a vector of length containing the coefficients au_i.
- is an integer vector of length corresponding to the permutation P.

– Function.

Compute the LQ factorization of , using the input matrix as a workspace. See also .

– Function.

Compute the LQ factorization of . See also .

– Function.

Perform an LQ factorization of such that . The default is to compute a thin factorization. The LQ factorization is the QR factorization of . is not extended with zeros if the full is requested.

– Function.

Compute the Bunch-Kaufman [3] factorization of a symmetric or Hermitian matrix and return a object. indicates which triangle of matrix to reference. If is , is assumed to be symmetric. If is , is assumed to be Hermitian. If is , rook pivoting is used. If is false, rook pivoting is not used. The following functions are available for objects: , , , , .

– Function.

is the same as , but saves space by overwriting the input , instead of creating a copy.

– Function.

Computes eigenvalues () and eigenvectors () of . See for details on the , , and arguments (for , , and matrices) and the and keyword arguments. The eigenvectors are returned columnwise.

Examples

is a wrapper around , extracting all parts of the factorization to a tuple; where possible, using is recommended.

Computes generalized eigenvalues () and vectors () of with respect to .

is a wrapper around , extracting all parts of the factorization to a tuple; where possible, using is recommended.

Examples

[3] J R Bunch and L Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Mathematics of Computation 31:137 (1977), 163-179. url.

– Function.

Returns the eigenvalues of .

For general non-symmetric matrices it is possible to specify how the matrix is balanced before the eigenvalue calculation. The option permutes the matrix to become closer to upper triangular, and scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is for both options.

Computes the generalized eigenvalues of and .

Examples

Returns the eigenvalues of . It is possible to calculate only a subset of the eigenvalues by specifying a covering indices of the sorted eigenvalues, e.g. the 2nd to 8th eigenvalues.

Returns the eigenvalues of . It is possible to calculate only a subset of the eigenvalues by specifying a pair and for the lower and upper boundaries of the eigenvalues.

– Function.

Same as , but saves space by overwriting the input , instead of creating a copy. The option permutes the matrix to become closer to upper triangular, and scales the matrix by its diagonal elements to make rows and columns more equal in norm.

Same as , but saves space by overwriting the input (and), instead of creating copies.

Same as , but saves space by overwriting the input , instead of creating a copy. is a range of eigenvalue *indices* to search for - for instance, the 2nd to 8th eigenvalues.

Same as , but saves space by overwriting the input , instead of creating a copy. is the lower bound of the interval to search for eigenvalues, and is the upper bound.

– Function.

Returns the largest eigenvalue of . The option permutes the matrix to become closer to upper triangular, and scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of are complex, this method will fail, since complex numbers cannot be sorted.

Examples

– Function.

Returns the smallest eigenvalue of . The option permutes the matrix to become closer to upper triangular, and scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of are complex, this method will fail, since complex numbers cannot be sorted.

Examples

– Function.

Returns a matrix whose columns are the eigenvectors of . (The th eigenvector can be obtained from the slice .)

If the optional vector of eigenvalues is specified, returns the specific corresponding eigenvectors.

Examples

Returns a matrix whose columns are the eigenvectors of . (The th eigenvector can be obtained from the slice .) The and keywords are the same as for .

Examples

Returns a matrix whose columns are the generalized eigenvectors of and . (The th eigenvector can be obtained from the slice .)

Examples

– Function.

Computes the eigenvalue decomposition of , returning an factorization object which contains the eigenvalues in and the eigenvectors in the columns of the matrix . (The th eigenvector can be obtained from the slice .)

The following functions are available for objects: , , and .

For general nonsymmetric matrices it is possible to specify how the matrix is balanced before the eigenvector calculation. The option permutes the matrix to become closer to upper triangular, and scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is for both options.

Examples

Computes the generalized eigenvalue decomposition of and , returning a factorization object which contains the generalized eigenvalues in and the generalized eigenvectors in the columns of the matrix . (The th generalized eigenvector can be obtained from the slice .)

Computes the eigenvalue decomposition of , returning an factorization object which contains the eigenvalues in and the eigenvectors in the columns of the matrix . (The th eigenvector can be obtained from the slice .)

The following functions are available for objects: , , and .

The specifies indices of the sorted eigenvalues to search for.

Note

If is not , where is the dimension of , then the returned factorization will be a *truncated* factorization.

Computes the eigenvalue decomposition of , returning an factorization object which contains the eigenvalues in and the eigenvectors in the columns of the matrix . (The th eigenvector can be obtained from the slice .)

The following functions are available for objects: , , and .

 is the lower bound of the window of eigenvalues to search for, and is the upper bound.

Note

If [,] does not contain all eigenvalues of , then the returned factorization will be a *truncated* factorization.

– Function.

Same as , but saves space by overwriting the input (and), instead of creating a copy.

– Function.

Compute the Hessenberg decomposition of and return a object. If is the factorization object, the unitary matrix can be accessed with and the Hessenberg matrix with . When is extracted, the resulting type is the object, and may be converted to a regular matrix with (or for short).

Examples

– Function.

is the same as , but saves space by overwriting the input , instead of creating a copy.

– Function.

Computes the Schur factorization of the matrix . The (quasi) triangular Schur factor can be obtained from the object with either or and the orthogonal/unitary Schur vectors can be obtained with or such that . The eigenvalues of can be obtained with .

Examples

Computes the Generalized Schur (or QZ) factorization of the matrices and . The (quasi) triangular Schur factors can be obtained from the object with and , the left unitary/orthogonal Schur vectors can be obtained with or and the right unitary/orthogonal Schur vectors can be obtained with or such that and . The generalized eigenvalues of and can be obtained with .

– Function.

Same as but uses the input argument as workspace.

Same as but uses the input matrices and as workspace.

– Function.

Computes the Schur factorization of the matrix . The methods return the (quasi) triangular Schur factor and the orthogonal/unitary Schur vectors such that . The eigenvalues of are returned in the vector .

See .

Examples

See .

– Function.

Reorders the Schur factorization of a matrix according to the logical array returning the reordered factorization object. The selected eigenvalues appear in the leading diagonal of and the corresponding leading columns of form an orthogonal/unitary basis of the corresponding right invariant subspace. In the real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via .

Reorders the Schur factorization of a real matrix according to the logical array returning the reordered matrices and as well as the vector of eigenvalues . The selected eigenvalues appear in the leading diagonal of and the corresponding leading columns of form an orthogonal/unitary basis of the corresponding right invariant subspace. In the real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via .

Reorders the Generalized Schur factorization of a matrix pair according to the logical array and returns a GeneralizedSchur object . The selected eigenvalues appear in the leading diagonal of both and , and the left and right orthogonal/unitary Schur vectors are also reordered such that still holds and the generalized eigenvalues of and can still be obtained with .

Reorders the Generalized Schur factorization of a matrix pair according to the logical array and returns the matrices , , , and vectors and . The selected eigenvalues appear in the leading diagonal of both and , and the left and right unitary/orthogonal Schur vectors are also reordered such that still holds and the generalized eigenvalues of and can still be obtained with .

– Function.

Same as but overwrites the factorization .

Same as but overwrites the input arguments.

Same as but overwrites the factorization .

Same as but overwrites the factorization the input arguments.

– Function.

Compute the singular value decomposition (SVD) of and return an object.

, , and can be obtained from the factorization with , , and , such that . The algorithm produces and hence is more efficient to extract than . The singular values in are sorted in descending order.

If (default), a thin SVD is returned. For a $M \times N$ matrix , is $M \times M$ for a full SVD () and $M \times \min(M, N)$ for a thin SVD.

Examples

Compute the generalized SVD of and , returning a factorization object , such that and .

For an M-by-N matrix and P-by-N matrix ,

- is a M-by-M orthogonal matrix,
- is a P-by-P orthogonal matrix,
- is a N-by-N orthogonal matrix,
- is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block triangular,
- is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,

is the effective numerical rank of the matrix .

The entries of and are related, as explained in the LAPACK documentation for the generalized SVD and the xGGSVD3 routine which is called underneath (in LAPACK 3.6.0 and newer).

– Function.

is the same as , but saves space by overwriting the input , instead of creating a copy.

is the same as , but modifies the arguments and in-place, instead of making copies.

– Function.

Computes the SVD of , returning , vector , and such that . The singular values in are sorted in descending order.

If (default), a thin SVD is returned. For a $M \times N$ matrix , is $M \times M$ for a full SVD () and $M \times \min(M, N)$ for a thin SVD.

is a wrapper around , extracting all parts of the factorization to a tuple. Direct use of is therefore more efficient.

Examples

Wrapper around extracting all parts of the factorization to a tuple. Direct use of is therefore generally more efficient. The function returns the generalized SVD of and , returning , , , , and such that and .

– Function.

Returns the singular values of in descending order.

Examples

Return the generalized singular values from the generalized singular value decomposition of and . See also .

– Type.

A Givens rotation linear operator. The fields and represent the cosine and sine of the rotation angle, respectively. The type supports left multiplication and conjugated transpose right multiplication . The type doesn't have a and can therefore be multiplied with matrices of arbitrary size as long as for or for .

See also:

– Function.

Computes the Givens rotation and scalar such that for any vector where

the result of the multiplication

has the property that

See also:

Computes the Givens rotation and scalar such that the result of the multiplication

has the property that

See also:

Computes the Givens rotation and scalar such that the result of the multiplication

has the property that

See also:

Computes the Givens rotation and scalar such that the result of the multiplication

has the property that

See also:

– Function.

Upper triangle of a matrix.

Examples

Returns the upper triangle of starting from the th superdiagonal.

Examples

– Function.

Upper triangle of a matrix, overwriting in the process. See also .

Returns the upper triangle of starting from the th superdiagonal, overwriting in the process.

Examples

– Function.

Lower triangle of a matrix.

Examples

Returns the lower triangle of starting from the th superdiagonal.

Examples

– Function.

Lower triangle of a matrix, overwriting in the process. See also .

Returns the lower triangle of starting from the th superdiagonal, overwriting in the process.

Examples

– Function.

A giving the indices of the th diagonal of the matrix .

Examples

```

```

– Function.

```

```

The th diagonal of a matrix, as a vector. Use to construct a diagonal matrix.

Examples

```

```

– Function.

```

```

Construct a matrix by placing on the th diagonal. This constructs a full matrix; if you want a storage-efficient version with fast arithmetic, use instead.

Examples

```

```

– Function.

```

```

Scale an array by a scalar overwriting in-place.

If is a matrix and is a vector, then scales each column of by (similar to), while scales each row of by (similar to), again operating in-place on . An exception is thrown if the scaling produces a number not representable by the element type of , e.g. for integer types.

Examples

– Function.

Compute the rank of a matrix by counting how many singular values of have magnitude greater than . By default, the value of is the largest dimension of multiplied by the of the of .

Examples

– Function.

Compute the -norm of a vector or the operator norm of a matrix , defaulting to the 2-norm.

For vectors, this is equivalent to and equal to:

$$\|A\|_p = \left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p}$$

with a_i the entries of A and n its length.

 can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, returns the largest value in , whereas returns the smallest.

Examples

For matrices, the matrix norm induced by the vector -norm is used, where valid values of are , , or . (Note that for sparse matrices, is currently not implemented.) Use to compute the Frobenius norm.

When , the matrix norm is the maximum absolute column sum of :

$$\|A\|_1 = \max_{1jn} \sum_{i=1}^{m} |a_{ij}|$$

with a_{ij} the entries of A, and m and n its dimensions.

When , the matrix norm is the spectral norm, equal to the largest singular value of .

When , the matrix norm is the maximum absolute row sum of :

$$\|A\|_\infty = \max_{1im} \sum_{j=1}^{n} |a_{ij}|$$

Examples

For numbers, return $\left(|x|^p\right)^{1/p}$. This is equivalent to .

For row vectors, return the q-norm of , which is equivalent to the p-norm with value . They coincide at .

The difference in norm between a vector space and its dual arises to preserve the relationship between duality and the inner product, and the result is consistent with the p-norm of matrix.

– Function.

For any iterable container (including arrays of any dimension) of numbers (or any element type for which is defined), compute the -norm (defaulting to) as if were a vector of the corresponding length.

The -norm is defined as:

$$\|A\|_p = \left(\sum_{i=1}^{n} |a_i|^p\right)^{1/p}$$

with a_i the entries of A and n its length.

 can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, returns the largest value in , whereas returns the smallest. If is a matrix and , then this is equivalent to the Frobenius norm.

Examples

For numbers, return $\left(|x|^p\right)^{1/p}$.

– Function.

Normalize the vector in-place so that its -norm equals unity, i.e. . See also and .

– Function.

Normalize the vector so that its -norm equals unity, i.e. . See also and .

Examples

– Function.

Condition number of the matrix , computed using the operator -norm. Valid values for are , (default), or .

– Function.

$$\kappa_S(M,p) = \left\| |M| \left| M^{-1} \right| \right\|_p$$
$$\kappa_S(M,x,p) = \left\| |M| \left| M^{-1} \right| |x| \right\|_p$$

Skeel condition number κ_S of the matrix , optionally with respect to the vector , as computed using the operator -norm. is by default, if not provided. Valid values for are , , or .

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

– Function.

Matrix trace. Sums the diagonal elements of .

Examples

– Function.

Matrix determinant.

Examples

– Function.

Log of matrix determinant. Equivalent to , but may provide increased accuracy and/or speed.

Examples

– Function.

Log of absolute value of matrix determinant. Equivalent to , but may provide increased accuracy and/or speed.

– Method.

Matrix inverse. Computes matrix such that , where is the identity matrix. Computed by solving the left-division .

Examples

– Function.

Computes the Moore-Penrose pseudoinverse.

For matrices with floating point elements, it is convenient to compute the pseudoinverse by inverting only singular values above a given threshold, .

The optimal choice of varies both with the value of and the intended application of the pseudoinverse. The default value of is , which is essentially machine epsilon for the real part of a matrix element multiplied by the larger matrix dimension. For inverting dense ill-conditioned matrices in a least-squares sense, is recommended.

For more information, see [4], [5], [6], [7].

Examples

– Function.

Basis for nullspace of .

Examples

– Function.

Construct a matrix by repeating the given matrix (or vector) times in dimension 1 and times in dimension 2.

Examples

[4] Issue 8859, "Fix least squares", https://github.com/JuliaLang/julia/pull/8859

[5] Åke Björck, "Numerical Methods for Least Squares Problems", SIAM Press, Philadelphia, 1996, "Other Titles in Applied Mathematics", Vol. 51. doi:10.1137/1.9781611971484

[6] G. W. Stewart, "Rank Degeneracy", SIAM Journal on Scientific and Statistical Computing, 5(2), 1984, 403-413. doi:10.1137/0905030

[7] Konstantinos Konstantinides and Kung Yao, "Statistical analysis of effective singular values in matrix rank determination", IEEE Transactions on Acoustics, Speech and Signal Processing, 36(5), 1988, 757-763. doi:10.1109/29.1585

– Function.

Kronecker tensor product of two vectors or two matrices.

Examples

– Function.

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

Examples

– Function.

Perform simple linear regression using Ordinary Least Squares. Returns and such that is the closest straight line to the given points , i.e., such that the squared error between and is minimized.

Examples

See also:

, , , .

– Function.

Compute the matrix exponential of , defined by

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

For symmetric or Hermitian , an eigendecomposition () is used, otherwise the scaling and squaring algorithm (see [8]) is chosen.

Examples

– Function.

If has no negative real eigenvalue, compute the principal matrix logarithm of , i.e. the unique matrix X such that $e^X = A$ and $-\pi < Im(\lambda) < \pi$ for all the eigenvalues λ of X. If has nonpositive eigenvalues, a nonprincipal matrix function is returned whenever possible.

If is symmetric or Hermitian, its eigendecomposition () is used, if is triangular an improved version of the inverse scaling and squaring method is employed (see [9] and [10]). For general matrices, the complex Schur form () is computed and the triangular algorithm is used on the triangular factor.

[8] Nicholas J. Higham, "The squaring and scaling method for the matrix exponential revisited", SIAM Journal on Matrix Analysis and Applications, 26(4), 2005, 1179-1193. doi:10.1137/090768539

[9] Awad H. Al-Mohy and Nicholas J. Higham, "Improved inverse scaling and squaring algorithms for the matrix logarithm", SIAM Journal on Scientific Computing, 34(4), 2012, C153-C169. doi:10.1137/110852553

[10] Awad H. Al-Mohy, Nicholas J. Higham and Samuel D. Relton, "Computing the Fréchet derivative of the matrix logarithm and estimating the condition number", SIAM Journal on Scientific Computing, 35(4), 2013, C394-C410. doi:10.1137/120885991

Examples

– Function.

If has no negative real eigenvalues, compute the principal matrix square root of , that is the unique matrix X with eigenvalues having positive real part such that $X^2 = A$. Otherwise, a nonprincipal square root is returned.

If is symmetric or Hermitian, its eigendecomposition () is used to compute the square root. Otherwise, the square root is determined by means of the Björck-Hammarling method [11], which computes the complex Schur form () and then the complex square root of the triangular factor.

Examples

– Function.

Computes the solution to the continuous Lyapunov equation , where no eigenvalue of has a zero real part and no two eigenvalues are negative complex conjugates of each other.

– Function.

Computes the solution to the Sylvester equation , where , and have compatible dimensions and and have no eigenvalues with equal real part.

[11] Åke Björck and Sven Hammarling, "A Schur method for the square root of a matrix", Linear Algebra and its Applications, 52-53, 1983, 127-140. doi:10.1016/0024-3795(83)80010-X

– Function.

Test that a factorization of a matrix succeeded.

– Function.

Test whether a matrix is symmetric.

Examples

– Function.

Test whether a matrix is positive definite by trying to perform a Cholesky factorization of . See also

Examples

– Function.

Test whether a matrix is positive definite by trying to perform a Cholesky factorization of , overwriting in the process. See also .

Examples

– Function.

Test whether a matrix is lower triangular.

Examples

– Function.

Test whether a matrix is upper triangular.

Examples

– Function.

Test whether a matrix is diagonal.

Examples

– Function.

Test whether a matrix is Hermitian.

Examples

```

```

– Type.

```

```

A lazy-view wrapper of an , which turns a length- vector into a shaped row vector and represents the transpose of a vector (the elements are also transposed recursively). This type is usually constructed (and unwrapped) via the function or operator (or related or operator).

By convention, a vector can be multiplied by a matrix on its left () whereas a row vector can be multiplied by a matrix on its right (such that). It differs from a -sized matrix by the facts that its transpose returns a vector and the inner product returns a scalar, but will otherwise behave similarly.

– Type.

```

```

A lazy-view wrapper of an , taking the elementwise complex conjugate. This type is usually constructed (and unwrapped) via the function (or related), but currently this is the default behavior for only. For other arrays, the constructor can be used directly.

Examples

```

```

– Function.

```

```

The transposition operator ().

Examples

The transposition operator ().

Examples

– Function.

Transpose array and store the result in the preallocated array , which should have a size corresponding to . No in-place transposition is supported and unexpected results will happen if and have overlapping memory regions.

– Function.

The conjugate transposition operator ().

Examples

– Function.

Conjugate transpose array and store the result in the preallocated array , which should have a size corresponding to . No in-place transposition is supported and unexpected results will happen if and have overlapping memory regions.

– Method.

Computes eigenvalues of using implicitly restarted Lanczos or Arnoldi iterations for real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

- : Number of eigenvalues
- : Number of Krylov vectors used in the computation; should satisfy for real symmetric problems and for other problems, where is the size of the input matrix . The default is . Note that these restrictions limit the input matrix to be of dimension at least 2.
- : type of eigenvalues to compute. See the note below.

	type of eigenvalues
	eigenvalues of largest magnitude (default)
	eigenvalues of smallest magnitude
	eigenvalues of largest real part
	eigenvalues of smallest real part
	eigenvalues of largest imaginary part (nonsymmetric or complex only)
	eigenvalues of smallest imaginary part (nonsymmetric or complex only)
	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric only)

- : parameter defining the relative tolerance for convergence of Ritz values (eigenvalue estimates). A Ritz value is considered converged when its associated residual is less than or equal to the product of and $max(^{2/3}, ||)$, where is LAPACK's machine epsilon. The residual associated with and its corresponding Ritz vector v is defined as the norm $||Av - v||$. The specified value of should be positive; otherwise, it is ignored and is used instead. Default: .
- : Maximum number of iterations (default = 300)
- : Specifies the level shift used in inverse iteration. If (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to using shift and invert iterations.
- : Returns the Ritz vectors (eigenvectors) if
- : starting vector from which to start the iterations

returns the requested eigenvalues in , the corresponding Ritz vectors (only if), the number of converged eigenvalues , the number of iterations and the number of matrix vector multiplications , as well as the final residual vector .

Examples

Note

The and keywords interact: the description of eigenvalues searched for by do *not* necessarily refer to the eigenvalues of , but rather the linear operator constructed by the specification of the iteration mode implied by .

	iteration mode	refers to eigenvalues of
	ordinary (forward)	A
real or complex	inverse with level shift	$(A-\sigma I)^{-1}$

Note

Although has a default value, the best choice depends strongly on the matrix . We recommend that users _always_ specify a value for which suits their specific needs.

For details of how the errors in the computed eigenvalues are estimated, see:

- B. N. Parlett, "The Symmetric Eigenvalue Problem", SIAM: Philadelphia, 2/e (1998), Ch. 13.2, "Accessing Accuracy in Lanczos Problems", pp. 290-292 ff.
- R. B. Lehoucq and D. C. Sorensen, "Deflation Techniques for an Implicitly Restarted Arnoldi Iteration", SIAM Journal on Matrix Analysis and Applications (1996), 17(4), 789–821. doi:10.1137/S0895479895281484

– Method.

Computes generalized eigenvalues of and using implicitly restarted Lanczos or Arnoldi iterations for real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

- : Number of eigenvalues
- : Number of Krylov vectors used in the computation; should satisfy for real symmetric problems and for other problems, where is the size of the input matrices and . The default is . Note that these restrictions limit the input matrix to be of dimension at least 2.
- : type of eigenvalues to compute. See the note below.
- : relative tolerance used in the convergence criterion for eigenvalues, similar to in the method for the ordinary eigenvalue problem, but effectively for the eigenvalues of $B^{-1}A$ instead of A. See the documentation for the ordinary eigenvalue problem in and the accompanying note about .
- : Maximum number of iterations (default = 300)
- : Specifies the level shift used in inverse iteration. If (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to using shift and invert iterations.

	type of eigenvalues
	eigenvalues of largest magnitude (default)
	eigenvalues of smallest magnitude
	eigenvalues of largest real part
	eigenvalues of smallest real part
	eigenvalues of largest imaginary part (nonsymmetric or complex only)
	eigenvalues of smallest imaginary part (nonsymmetric or complex only)
	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric only)

- : Returns the Ritz vectors (eigenvectors) if
- : starting vector from which to start the iterations

returns the requested eigenvalues in , the corresponding Ritz vectors (only if), the number of converged eigenvalues , the number of iterations and the number of matrix vector multiplications , as well as the final residual vector .

Examples

Note

The and keywords interact: the description of eigenvalues searched for by do *not* necessarily refer to the eigenvalue problem $Av = Bv\lambda$, but rather the linear operator constructed by the specification of the iteration mode implied by .

	iteration mode	refers to the problem
	ordinary (forward)	$Av = Bv\lambda$
real or complex	inverse with level shift	$(A - \sigma B)^{-1}B = v\nu$

– Function.

Computes the largest singular values of using implicitly restarted Lanczos iterations derived from .

Inputs

- : Linear operator whose singular values are desired. may be represented as a subtype of , e.g., a sparse matrix, or any other type supporting the four methods , , , and .
- : Number of singular values. Default: 6.
- : If , return the left and right singular vectors and . If , omit the singular vectors. Default: .

- : tolerance, see .
- : Maximum number of iterations, see . Default: 1000.
- : Maximum size of the Krylov subspace, see (there called). Default: .
- : Initial guess for the first Krylov vector. It may have length , or 0.

Outputs

- : An object containing the left singular vectors, the requested values, and the right singular vectors. If , the left and right singular vectors will be empty.
- : Number of converged singular values.
- : Number of iterations.
- : Number of matrix–vector products used.
- : Final residual vector.

Examples

Implementation

is formally equivalent to calling to perform implicitly restarted Lanczos tridiagonalization on the Hermitian matrix $A'A$ or AA' such that the size is smallest.

– Function.

computes the peak flop rate of the computer by using double precision . By default, if no arguments are specified, it multiplies a matrix of size , where . If the underlying BLAS is using multiple threads, higher flop rates are realized. The number of BLAS threads can be set with .

If the keyword argument is set to , is run in parallel on all the worker processors. The flop rate of the entire parallel computer is returned. When running in parallel, only 1 BLAS thread is used. The argument still refers to the size of the problem that is solved on each processor.

53.2 Low-level matrix operations

Matrix operations involving transpositions operations like are converted by the Julia parser into calls to specially named functions like . If you want to overload these operations for your own types, then it is useful to know the names of these functions.

Also, in many cases there are in-place versions of matrix operations that allow you to supply a pre-allocated output vector or matrix. This is useful when optimizing critical code in order to avoid the overhead of repeated allocations. These in-place operations are suffixed with below (e.g.) according to the usual Julia convention.

– Function.

Compute in-place and store the result in , returning the result. If only two arguments are passed, then overwrites with the result.

The argument should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by or). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g.,), and performance-critical situations requiring usually also require fine-grained control over the factorization of .

– Function.

For matrices or vectors A and B, calculates $A \setminus B$.

– Function.

For matrices or vectors A and B, calculates $A \setminus B$.

– Function.

Calculates the matrix-matrix or matrix-vector product AB and stores the result in , overwriting the existing value of . Note that must not be aliased with either or .

Examples

– Function.

For matrices or vectors A and B, calculates AB.

– Function.

For matrices or vectors A and B, calculates AB.

– Function.

For matrices or vectors A and B, calculates A/B.

– Function.

For matrices or vectors A and B, calculates A/B.

– Function.

For matrices or vectors A and B, calculates $A \setminus B$.

– Function.

Similar to , but return $A \setminus B$, computing the result in-place in (or overwriting if is not supplied).

– Function.

For matrices or vectors A and B, calculates $A \setminus B$.

– Function.

For matrices or vectors A and B, calculates AB.

– Function.

For matrices or vectors A and B, calculates AB.

– Function.

For matrices or vectors A and B, calculates A/B.

– Function.

For matrices or vectors A and B, calculates A/B.

– Function.

For matrices or vectors A and B, calculates $A \setminus B$.

– Function.

Similar to , but return $A \setminus B$, computing the result in-place in (or overwriting if is not supplied).

– Function.

For matrices or vectors A and B, calculates $A \setminus B$.

– Function.

For matrices or vectors A and B, calculates AB.

– Function.

For matrices or vectors A and B, calculates AB.

– Function.

For matrices or vectors A and B, calculates A/B.

– Function.

For matrices or vectors A and B, calculates A/B.

53.3 BLAS Functions

In Julia (as in much of scientific computation), dense linear-algebra operations are based on the LAPACK library, which in turn is built on top of basic linear-algebra building-blocks known as the BLAS. There are highly optimized implementations of BLAS available for every computer architecture, and sometimes in high-performance linear algebra routines it is useful to call the BLAS functions directly.

provides wrappers for some of the BLAS functions. Those BLAS functions that overwrite one of the input arrays have names ending in . Usually, a BLAS function has four methods defined, for , , , and arrays.

BLAS Character Arguments

Many BLAS functions accept arguments that determine whether to transpose an argument (), which triangle of a matrix to reference (or), whether the diagonal of a triangular matrix can be assumed to be all ones () or which side of a matrix multiplication the input argument belongs on (). The possiblities are:

Multplication Order

	Meaning
	The argument goes on the *left* side of a matrix-matrix operation.
	The argument goes on the *right* side of a matrix-matrix operation.

Triangle Referencing

/	Meaning
	Only the *upper* triangle of the matrix will be used.
	Only the *lower* triangle of the matrix will be used.

Transposition Operation

/	Meaning
	The input matrix is not transposed or conjugated.
	The input matrix will be transposed.
	The input matrix will be conjugated and transposed.

Unit Diagonal

/	Meaning
	The diagonal values of the matrix will be read.
	The diagonal of the matrix is assumed to be all ones.

– Function.

Dot function for two complex vectors consisting of elements of array with stride and elements of array with stride .

Examples

– Function.

Dot function for two complex vectors, consisting of elements of array with stride and elements of array with stride , conjugating the first vector.

Examples

– Function.

Copy elements of array with stride to array with stride . Returns .

– Function.

2-norm of a vector consisting of elements of array with stride .

Examples

– Function.

Sum of the absolute values of the first elements of array with stride .

Examples

```
```

– Function.

```
```

Overwrite with , where is a scalar. Returns .

Examples

```
```

– Function.

```
```

Overwrite with for the first elements of array with stride . Returns .

– Function.

```
```

Returns scaled by for the first elements of array with stride .

– Function.

```
```

Rank-1 update of the matrix with vectors and as .

– Function.

```
```

Rank-1 update of the symmetric matrix with vector as . controls which triangle of is updated. Returns .

– Function.

Rank-k update of the symmetric matrix as or according to . Only the triangle of is used. Returns .

– Function.

Returns either the upper triangle or the lower triangle of , according to , of or , according to .

– Function.

Methods for complex arrays only. Rank-1 update of the Hermitian matrix with vector as . controls which triangle of is updated. Returns .

– Function.

Methods for complex arrays only. Rank-k update of the Hermitian matrix as or according to . Only the triangle of is updated. Returns .

– Function.

Methods for complex arrays only. Returns the triangle of or , according to .

– Function.

Update vector as or according to . The matrix is a general band matrix of dimension by with sub-diagonals and super-diagonals. and are scalars. Returns the updated .

– Function.

Returns or according to . The matrix is a general band matrix of dimension by with sub-diagonals and super-diagonals, and is a scalar.

– Function.

Update vector as where is a a symmetric band matrix of order with super-diagonals stored in the argument . The storage layout for is described the reference BLAS module, level-2 BLAS at http://www.netlib.org/lapack/explore-html/. Only the triangle of is used.

Returns the updated .

– Method.

Returns where is a symmetric band matrix of order with super-diagonals stored in the argument . Only the triangle of is used.

– Method.

Returns where is a symmetric band matrix of order with super-diagonals stored in the argument . Only the triangle of is used.

– Function.

Update as or the other three variants according to and . Returns the updated .

– Method.

Returns or the other three variants according to and .

– Method.

Returns or the other three variants according to and .

– Function.

Update the vector as or according to . and are scalars. Returns the updated .

– Method.

Returns or according to . is a scalar.

– Method.

Returns or according to .

– Function.

Update as or according to . is assumed to be symmetric. Only the triangle of is used. Returns the updated .

– Method.

Returns or according to . is assumed to be symmetric. Only the triangle of is used.

– Method.

Returns or according to . is assumed to be symmetric. Only the triangle of is used.

– Function.

Update the vector as . is assumed to be symmetric. Only the triangle of is used. and are scalars. Returns the updated .

– Method.

Returns . is assumed to be symmetric. Only the triangle of is used. is a scalar.

– Method.

Returns . is assumed to be symmetric. Only the triangle of is used.

– Function.

Update as or one of the other three variants determined by and . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones. Returns the updated .

– Function.

Returns or one of the other three variants determined by and . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones.

– Function.

Overwrite with the solution to or one of the other three variants determined by and . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones. Returns the updated .

– Function.

Returns the solution to or one of the other three variants determined by determined by and . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones.

– Function.

Returns , where is determined by . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones. The multiplication occurs in-place on .

– Function.

Returns , where is determined by . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones.

– Function.

Overwrite with the solution to or one of the other two variants determined by and . determines if the diagonal values are read or are assumed to be all ones. Returns the updated .

– Function.

Returns the solution to or one of the other two variants determined by and . determines if the diagonal values are read or are assumed to be all ones.

– Function.

Set the number of threads the BLAS library should use.

– Constant.

An object of type , representing an identity matrix of any size.

Examples

53.4 LAPACK Functions

provides wrappers for some of the LAPACK functions for linear algebra. Those functions that overwrite one of the input arrays have names ending in .

Usually a function has 4 methods defined, one each for , , and arrays.

Note that the LAPACK API provided by Julia can and will change in the future. Since this API is not user-facing, there is no commitment to support/deprecate this specific set of functions in future releases.

– Function.

Compute the LU factorization of a banded matrix . is the first subdiagonal containing a nonzero band, is the last superdiagonal containing one, and is the first dimension of the matrix . Returns the LU factorization in-place and , the vector of pivots used.

– Function.

Solve the equation . determines the orientation of . It may be (no transpose), (transpose), or (conjugate transpose). is the first subdiagonal containing a nonzero band, is the last superdiagonal containing one, and is the first dimension of the matrix . is the vector of pivots returned from . Returns the vector or matrix , overwriting in-place.

– Function.

Balance the matrix before computing its eigensystem or Schur factorization. can be one of (will not be permuted or scaled), (will only be permuted), (will only be scaled), or (will be both permuted and scaled). Modifies in-place and returns , , and . If permuting was turned on, if and or . contains information about the scaling/permutations performed.

– Function.

Transform the eigenvectors of a matrix balanced using to the unscaled/unpermuted eigenvectors of the original matrix. Modifies in-place. can be (left eigenvectors are transformed) or (right eigenvectors are transformed).

– Function.

Reduce in-place to bidiagonal form . Returns , containing the bidiagonal matrix ; , containing the diagonal elements of ; , containing the off-diagonal elements of ; , containing the elementary reflectors representing ; and , containing the elementary reflectors representing .

– Function.

Compute the factorization of , . contains scalars which parameterize the elementary reflectors of the factorization. must have length greater than or equal to the smallest dimension of .

Returns and modified in-place.

Compute the factorization of , .

Returns , modified in-place, and , which contains scalars which parameterize the elementary reflectors of the factorization.

– Function.

Compute the factorization of , . contains scalars which parameterize the elementary reflectors of the factorization. must have length greater than or equal to the smallest dimension of .

Returns and modified in-place.

Compute the factorization of , .

Returns , modified in-place, and , which contains scalars which parameterize the elementary reflectors of the factorization.

– Function.

Compute the factorization of , . contains scalars which parameterize the elementary reflectors of the factorization. must have length greater than or equal to the smallest dimension of .

Returns and modified in-place.

Compute the factorization of , .

Returns , modified in-place, and , which contains scalars which parameterize the elementary reflectors of the factorization.

– Function.

Compute the pivoted factorization of , using BLAS level 3. is a pivoting matrix, represented by . stores the elementary reflectors. must have length length greater than or equal to if is an matrix. must have length greater than or equal to the smallest dimension of .

, , and are modified in-place.

Compute the pivoted factorization of , using BLAS level 3. is a pivoting matrix, represented by . must have length greater than or equal to if is an matrix.

Returns and , modified in-place, and , which stores the elementary reflectors.

Compute the pivoted factorization of , using BLAS level 3.

Returns , modified in-place, , which represents the pivoting matrix , and , which stores the elementary reflectors.

– Function.

Compute the factorization of , . contains scalars which parameterize the elementary reflectors of the factorization. must have length greater than or equal to the smallest dimension of .

Returns and modified in-place.

Compute the factorization of , .

Returns , modified in-place, and , which contains scalars which parameterize the elementary reflectors of the factorization.

– Function.

Compute the blocked factorization of , . contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of sets the block size and it must be between 1 and . The second dimension of must equal the smallest dimension of .

Returns and modified in-place.

Compute the blocked factorization of , . sets the block size and it must be between 1 and , the second dimension of .

Returns , modified in-place, and , which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

– Function.

Recursively computes the blocked factorization of , . contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of sets the block size and it must be between 1 and . The second dimension of must equal the smallest dimension of .

Returns and modified in-place.

Recursively computes the blocked factorization of , .

Returns , modified in-place, and , which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

– Function.

Compute the pivoted factorization of , .

Returns , modified in-place, , the pivoting information, and an code which indicates success (), a singular value in (, in which case is singular), or an error code ().

– Function.

Transforms the upper trapezoidal matrix to upper triangular form in-place. Returns and , the scalar parameters for the elementary reflectors of the transformation.

– Function.

Multiplies the matrix by from the transformation supplied by . Depending on or the multiplication can be left-sided () or right-sided () and can be unmodified (), transposed (), or conjugate transposed (). Returns matrix which is modified in-place with the result of the multiplication.

– Function.

Solves the linear equation , , or using a QR or LQ factorization. Modifies the matrix/vector in place with the solution. is overwritten with its or factorization. may be one of (no modification), (transpose), or (conjugate transpose). searches for the minimum norm/least squares solution. may be under or over determined. The solution is returned in .

– Function.

Solves the linear equation where is a square matrix using the factorization of . is overwritten with its factorization and is overwritten with the solution . contains the pivoting information for the factorization of .

– Function.

Solves the linear equation , , or for square . Modifies the matrix/vector in place with the solution. is the factorization from , with the pivoting information. may be one of (no modification), (transpose), or (conjugate transpose).

– Function.

Computes the inverse of , using its factorization found by . is the pivot information output and contains the factorization of . is overwritten with its inverse.

– Function.

Solves the linear equation (), (), or () using the factorization of . may be , in which case will be equilibrated and copied to ; , in which case and from a previous factorization are inputs; or , in which case will be copied to and then factored. If , may be , meaning has not been equilibrated; , meaning was multiplied by from the left; , meaning was multiplied by from the right; or , meaning was multiplied by from the left and from the right. If and or the elements of must all be positive. If and or the elements of must all be positive.

Returns the solution ; , which is an output if is not , and describes the equilibration that was performed; , the row equilibration diagonal; , the column equilibration diagonal; , which may be overwritten with its equilibrated form (if and) or (if and); , the reciprocal condition number of after equilbrating; , the forward error bound for each solution vector in ; , the forward error bound for each solution vector in ; and , the reciprocal pivot growth factor.

The no-equilibration, no-transpose simplification of .

– Function.

Computes the least norm solution of by finding the factorization of , then dividing-and-conquering the problem. is overwritten with the solution . Singular values below will be treated as zero. Returns the solution in and the effective rank of in .

– Function.

Computes the least norm solution of by finding the full factorization of , then dividing-and-conquering the problem. is overwritten with the solution . Singular values below will be treated as zero. Returns the solution in and the effective rank of in .

– Function.

Solves the equation where is subject to the equality constraint . Uses the formula to solve. Returns and the residual sum-of-squares.

– Function.

Finds the eigensystem of . If , the left eigenvectors of aren't computed. If , the right eigenvectors of aren't computed. If or , the corresponding eigenvectors are computed. Returns the eigenvalues in , the right eigenvectors in , and the left eigenvectors in .

– Function.

Finds the singular value decomposition of , , using a divide and conquer approach. If , all the columns of and the rows of are computed. If , no columns of or rows of are computed. If , is overwritten with the columns of (thin) and the rows of (thin) . If , the columns of (thin) and the rows of (thin) are computed and returned separately.

– Function.

Finds the singular value decomposition of , . If , all the columns of are computed. If all the rows of are computed. If , no columns of are computed. If no rows of are computed. If , is overwritten with the columns of (thin) . If , is overwritten with the rows of (thin) . If , the columns of (thin) are computed and returned separately. If the rows of (thin) are computed and returned separately. and can't both be .

Returns , , and , where are the singular values of .

– Function.

Finds the generalized singular value decomposition of and , and . has on its diagonal and has on its diagonal. If , the orthogonal/unitary matrix is computed. If the orthogonal/unitary matrix is computed. If , the orthogonal/unitary matrix is computed. If , or is , that matrix is not computed. This function is only available in LAPACK versions prior to 3.6.0.

– Function.

Finds the generalized singular value decomposition of and , and . has on its diagonal and has on its diagonal. If , the orthogonal/unitary matrix is computed. If the orthogonal/unitary matrix is computed. If , the orthogonal/unitary matrix is computed. If , , or is , that matrix is not computed. This function requires LAPACK 3.6.0.

– Function.

Finds the eigensystem of with matrix balancing. If , the left eigenvectors of aren't computed. If , the right eigenvectors of aren't computed. If or , the corresponding eigenvectors are computed. If , no balancing is performed.

If , is permuted but not scaled. If , is scaled but not permuted. If , is permuted and scaled. If , no reciprocal condition numbers are computed. If , reciprocal condition numbers are computed for the eigenvalues only. If , reciprocal condition numbers are computed for the right eigenvectors only. If , reciprocal condition numbers are computed for the right eigenvectors and the eigenvectors. If , the right and left eigenvectors must be computed.

– Function.

Finds the generalized eigendecomposition of and . If , the left eigenvectors aren't computed. If , the right eigenvectors aren't computed. If or , the corresponding eigenvectors are computed.

– Function.

Solves the equation where is a tridiagonal matrix with on the subdiagonal, on the diagonal, and on the superdiagonal.

Overwrites with the solution and returns it.

– Function.

Finds the factorization of a tridiagonal matrix with on the subdiagonal, on the diagonal, and on the superdiagonal.

Modifies , , and in-place and returns them and the second superdiagonal and the pivoting vector .

– Function.

Solves the equation (), (), or () using the factorization computed by . is overwritten with the solution .

– Function.

Explicitly finds the matrix of a factorization after calling on . Uses the output of . is overwritten by .

– Function.

Explicitly finds the matrix of a factorization after calling on . Uses the output of . is overwritten by .

– Function.

Explicitly finds the matrix of a factorization after calling on . Uses the output of . is overwritten by .

– Function.

Explicitly finds the matrix of a factorization after calling on . Uses the output of . is overwritten by .

– Function.

Computes (), (), () for or the equivalent right-sided multiplication for using from a factorization of computed using . is overwritten.

– Function.

Computes (), (), () for or the equivalent right-sided multiplication for using from a factorization of computed using . is overwritten.

– Function.

Computes (), (), () for or the equivalent right-sided multiplication for using from a factorization of computed using . is overwritten.

– Function.

Computes (), (), () for or the equivalent right-sided multiplication for using from a factorization of computed using . is overwritten.

– Function.

Computes (), (), () for or the equivalent right-sided multiplication for using from a factorization of computed using . is overwritten.

– Function.

Finds the solution to where is a symmetric or Hermitian positive definite matrix. If the upper Cholesky decomposition of is computed. If the lower Cholesky decomposition of is computed. is overwritten by its Cholesky decomposition. is overwritten with the solution .

– Function.

Computes the Cholesky (upper if , lower if) decomposition of positive-definite matrix . is overwritten and returned with an info code.

– Function.

Computes the inverse of positive-definite matrix after calling to find its (upper if , lower if) Cholesky decomposition.

is overwritten by its inverse and returned.

– Function.

Finds the solution to where is a symmetric or Hermitian positive definite matrix whose Cholesky decomposition was computed by . If the upper Cholesky decomposition of was computed. If the lower Cholesky decomposition of was computed. is overwritten with the solution .

– Function.

Computes the (upper if , lower if) pivoted Cholesky decomposition of positive-definite matrix with a user-set tolerance . is overwritten by its Cholesky decomposition.

Returns , the pivots , the rank of , and an code. If , the factorization succeeded. If , then is indefinite or rank-deficient.

– Function.

Solves for positive-definite tridiagonal . is the diagonal of and is the off-diagonal. is overwritten with the solution and returned.

– Function.

Computes the LDLt factorization of a positive-definite tridiagonal matrix with as diagonal and as off-diagonal. and are overwritten and returned.

– Function.

Solves for positive-definite tridiagonal with diagonal and off-diagonal after computing 's LDLt factorization using . is overwritten with the solution .

– Function.

Finds the inverse of (upper if , lower if) triangular matrix . If , has non-unit diagonal elements. If , all diagonal elements of are one. is overwritten with its inverse.

– Function.

Solves (), (), or () for (upper if , lower if) triangular matrix . If , has non-unit diagonal elements. If , all diagonal elements of are one. is overwritten with the solution .

– Function.

Finds the reciprocal condition number of (upper if , lower if) triangular matrix . If , has non-unit diagonal elements. If , all diagonal elements of are one. If , the condition number is found in the infinity norm. If or , the condition number is found in the one norm.

– Function.

Finds the eigensystem of an upper triangular matrix . If , the right eigenvectors are computed. If , the left eigenvectors are computed. If , both sets are computed. If , all eigenvectors are found. If , all eigenvectors are found and backtransformed using and . If , only the eigenvectors corresponding to the values in are computed.

– Function.

Estimates the error in the solution to (), (), () for , or the equivalent equations a right-handed after computing using . If , is upper triangular. If , is lower triangular. If , has non-unit diagonal elements. If , all diagonal elements of are one. and are optional inputs. is the forward error and is the backward error, each component-wise.

– Function.

Computes the eigensystem for a symmetric tridiagonal matrix with as diagonal and as off-diagonal. If only the eigenvalues are found and returned in . If then the eigenvectors are also found and returned in .

– Function.

Computes the eigenvalues for a symmetric tridiagonal matrix with as diagonal and as off-diagonal. If , all the eigenvalues are found. If , the eigenvalues in the half-open interval are found. If , the eigenvalues with indices between and are found. If , eigvalues are ordered within a block. If , they are ordered across all the blocks. can be set as a tolerance for convergence.

– Function.

Computes the eigenvalues () or eigenvalues and eigenvectors () for a symmetric tridiagonal matrix with as diagonal and as off-diagonal. If , all the eigenvalues are found. If , the eigenvalues in the half-open interval are found. If , the eigenvalues with indices between and are found. The eigenvalues are returned in and the eigenvectors in .

– Function.

Computes the eigenvectors for a symmetric tridiagonal matrix with as diagonal and as off-diagonal. specifies the input eigenvalues for which to find corresponding eigenvectors. specifies the submatrices corresponding to the eigenvalues in . specifies the splitting points between the submatrix blocks.

– Function.

Converts a symmetric matrix (which has been factorized into a triangular matrix) into two matrices and . If , is upper triangular. If , it is lower triangular. is the pivot vector from the triangular factorization. is overwritten by and .

– Function.

Finds the solution to for symmetric matrix . If , the upper half of is stored. If , the lower half is stored. is overwritten by the solution . is overwritten by its Bunch-Kaufman factorization. contains pivoting information about the factorization.

– Function.

Computes the Bunch-Kaufman factorization of a symmetric matrix . If , the upper half of is stored. If , the lower half is stored.

Returns , overwritten by the factorization, a pivot vector , and the error code which is a non-negative integer. If is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position .

– Function.

Computes the inverse of a symmetric matrix using the results of . If , the upper half of is stored. If , the lower half is stored. is overwritten by its inverse.

– Function.

Solves the equation for a symmetric matrix using the results of . If , the upper half of is stored. If , the lower half is stored. is overwritten by the solution .

– Function.

Finds the solution to for Hermitian matrix . If , the upper half of is stored. If , the lower half is stored. is overwritten by the solution . is overwritten by its Bunch-Kaufman factorization. contains pivoting information about the factorization.

– Function.

Computes the Bunch-Kaufman factorization of a Hermitian matrix . If , the upper half of is stored. If , the lower half is stored.

Returns , overwritten by the factorization, a pivot vector , and the error code which is a non-negative integer. If is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position .

– Function.

|

Computes the inverse of a Hermitian matrix using the results of . If , the upper half of is stored. If , the lower half is stored. is overwritten by its inverse.

– Function.

|

Solves the equation for a Hermitian matrix using the results of . If , the upper half of is stored. If , the lower half is stored. is overwritten by the solution .

– Function.

|

Finds the eigenvalues () or eigenvalues and eigenvectors () of a symmetric matrix . If , the upper triangle of is used. If , the lower triangle of is used.

– Function.

|

Finds the eigenvalues () or eigenvalues and eigenvectors () of a symmetric matrix . If , the upper triangle of is used. If , the lower triangle of is used. If , all the eigenvalues are found. If , the eigenvalues in the half-open interval are found. If , the eigenvalues with indices between and are found. can be set as a tolerance for convergence.

The eigenvalues are returned in and the eigenvectors in .

– Function.

|

Finds the generalized eigenvalues () or eigenvalues and eigenvectors () of a symmetric matrix and symmetric positive-definite matrix . If , the upper triangles of and are used. If , the lower triangles of and are used. If , the problem to solve is . If , the problem to solve is . If , the problem to solve is .

– Function.

|

Computes the singular value decomposition of a bidiagonal matrix with on the diagonal and on the off-diagonal. If , is the superdiagonal. If , is the subdiagonal. Can optionally also compute the product .

Returns the singular values in , and the matrix overwritten with .

– Function.

|

Computes the singular value decomposition of a bidiagonal matrix with on the diagonal and on the off-diagonal using a divide and conqueq method. If , is the superdiagonal. If , is the subdiagonal. If , only the singular values are found. If , the singular values and vectors are found. If , the singular values and vectors are found in compact form. Only works for real types.

Returns the singular values in , and if , the compact singular vectors in .

– Function.

|

Finds the reciprocal condition number of matrix . If , the condition number is found in the infinity norm. If or , the condition number is found in the one norm. must be the result of and is the norm of in the relevant norm.

– Function.

|

Converts a matrix to Hessenberg form. If is balanced with then and are the outputs of . Otherwise they should be and . contains the elementary reflectors of the factorization.

– Function.

|

Explicitly finds , the orthogonal/unitary matrix from . , , , and must correspond to the input/output to .

– Function.

|

Computes the eigenvalues () or the eigenvalues and Schur vectors () of matrix . is overwritten by its Schur form.

Returns , containing the Schur vectors, and , containing the eigenvalues.

– Function.

|

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (), or right Schur vectors () of and .

The generalized eigenvalues are returned in and . The left Schur vectors are returned in and the right Schur vectors are returned in .

– Function.

|

Reorder the Schur factorization of a matrix. If , the Schur vectors are reordered. If they are not modified. and specify the reordering of the vectors.

– Function.

|

Reorder the Schur factorization of a matrix and optionally finds reciprocal condition numbers. If , no condition numbers are found. If , only the condition number for this cluster of eigenvalues is found. If , only the condition number for the invariant subspace is found. If then the condition numbers for the cluster and subspace are found. If the Schur vectors are updated. If the Schur vectors are not modified. determines which eigenvalues are in the cluster.

Returns , , and reordered eigenvalues in .

– Function.

|

Reorders the vectors of a generalized Schur decomposition. specifices the eigenvalues in each cluster.

– Function.

|

Solves the Sylvester matrix equation where and are both quasi-upper triangular. If , is not modified. If , is transposed. If , is conjugate transposed. Similarly for and . If , the equation is solved. If , the equation is solved.

Returns (overwriting) and .

Chapter 54

Constants

– Constant.

The singleton instance of type , used by convention when there is no value to return (as in a C function). Can be converted to an empty value.

– Constant.

A string containing the script name passed to Julia from the command line. Note that the script name remains unchanged from within included files. Alternatively see .

– Constant.

An array of the command line arguments passed to Julia, as strings.

– Constant.

The C null pointer constant, sometimes used when calling external code.

– Constant.

A object describing which version of Julia is in use. For details see Version Number Literals.

– Constant.

An array of paths as strings or custom loader objects for the function and and statements to consider when loading code. To create a custom loader type, define the type and then add appropriate methods to the function with the following signature:

The argument is the current value in , is the name of the module to load, and is the path of any previously found code to provide . If no provider has been found earlier in then the value of will be . Custom loader functionality is experimental and may break or change in Julia 1.0.

– Constant.

A string containing the full path to the directory containing the executable.

– Constant.

The number of logical CPU cores available in the system.

See the Hwloc.jl package for extended information, including number of physical cores.

– Constant.

Standard word size on the current machine, in bits.

– Constant.

A symbol representing the name of the operating system, as returned by of the build configuration.

– Constant.

A symbol representing the architecture of the build configuration.

– Constant.

A string containing the build triple.

See also:

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

Chapter 55

Filesystem

– Function.

Get the current working directory.

– Method.

Set the current working directory.

– Method.

Temporarily changes the current working directory and applies function before returning.

– Function.

Returns the files and directories in the directory (or the current working directory if not given).

– Function.

The method returns an iterator that walks the directory tree of a directory. The iterator returns a tuple containing . The directory tree can be traversed top-down or bottom-up. If encounters a it will rethrow the error by default. A custom error handling function can be provided through keyword argument. is called with a as argument.

– Function.

Make a new directory with name and permissions . defaults to , modified by the current file creation mask. This function never creates more than one directory. If the directory already exists, or some intermediate directories do not exist, this function throws an error. See for a function which creates all required intermediate directories.

– Function.

Create all directories in the given , with permissions . defaults to , modified by the current file creation mask.

– Function.

Creates a symbolic link to with the name .

Note

This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

– Function.

Returns the target location a symbolic link points to.

– Function.

Change the permissions mode of to . Only integer s (e.g.) are currently supported. If and the path is a directory all permissions in that directory will be recursively changed.

– Function.

Change the owner and/or group of to and/or . If the value entered for or is the corresponding ID will not change. Only integer s and s are currently supported.

– Function.

Returns a structure whose fields contain information about the file. The fields of the structure are:

Name	Description
size	The size (in bytes) of the file
device	ID of the device that contains the file
inode	The inode number of the file
mode	The protection mode of the file
nlink	The number of hard links to the file
uid	The user id of the owner of the file
gid	The group id of the file owner
rdev	If this file refers to a device, the ID of the device it refers to
blksize	The file-system preferred block size for the file
blocks	The number of such blocks allocated
mtime	Unix timestamp of when the file was last modified
ctime	Unix timestamp of when the file was created

– Function.

Like , but for symbolic links gets the info for the link itself rather than the file it refers to. This function must be called on a file path rather than a file object or a file descriptor.

– Function.

Equivalent to

– Function.

Equivalent to .

– Function.

Equivalent to

– Function.

Equivalent to .

– Function.

Gets the permissions of the owner of the file as a bitfield of

Value	Description
01	Execute Permission
02	Write Permission
04	Read Permission

For allowed arguments, see .

– Function.

Like but gets the permissions of the group owning the file.

– Function.

Like but gets the permissions for people who neither own the file nor are a member of the group owning the file

– Function.

Copy the file, link, or directory from to . will first remove an existing .

If , and is a symbolic link, will be created as a symbolic link. If and is a symbolic link, will be a copy of the file or directory refers to.

– Function.

Download a file from the given url, optionally renaming it to the given local file name. Note that this function relies on the availability of external tools such as , or to download the file and is provided for convenience. For production use or situations in which more options are needed, please use a package that provides the desired functionality instead.

– Function.

Move the file, link, or directory from to . will first remove an existing .

– Function.

Delete the file, link, or empty directory at the given path. If is passed, a non-existing path is not treated as error. If is passed and the path is a directory, then all contents are removed recursively.

– Function.

Update the last-modified timestamp on a file to the current time.

– Function.

Generate a unique temporary file path.

– Function.

Obtain the path of a temporary directory (possibly shared with other processes).

– Method.

Returns , where is the path of a new temporary file in and is an open file object for this path.

– Method.

Apply the function to the result of and remove the temporary file upon completion.

– Method.

Create a temporary directory in the directory and return its path. If does not exist, throw an error.

– Method.

Apply the function to the result of and remove the temporary directory upon completion.

– Function.

Returns if is a block device, otherwise.

– Function.

Returns if is a character device, otherwise.

– Function.

Returns if is a directory, otherwise.

– Function.

Returns if is a FIFO, otherwise.

– Function.

Returns if is a regular file, otherwise.

– Function.

Returns if is a symbolic link, otherwise.

– Function.

Returns if is a mount point, otherwise.

– Function.

Returns if is a valid filesystem path, otherwise.

– Function.

Returns if has the setgid flag set, otherwise.

– Function.

Returns if has the setuid flag set, otherwise.

– Function.

Returns if is a socket, otherwise.

– Function.

Returns if has the sticky bit set, otherwise.

– Function.

Return the current user's home directory.

Note
determines the home directory via 's . For details (for example on how to specify the home directory via environment variables), see the documentation.

– Function.

Get the directory part of a path.

See also:

– Function.

Get the file name part of a path.

See also:

– Macro.

expands to a string with the path to the file containing the macrocall, or an empty string if evaluated by . Returns if the macro was missing parser source information. Alternatively see .

– Macro.

expands to a string with the absolute path to the directory of the file containing the macrocall. Returns the current working directory if run from a REPL or if evaluated by .

– Macro.

expands to the line number of the location of the macrocall. Returns if the line number could not be determined.

– Function.

Determines whether a path is absolute (begins at the root directory).

– Function.

Determines whether a path refers to a directory (for example, ends with a path separator).

– Function.

Join path components into a full path. If some argument is an absolute path, then prior components are dropped.

– Function.

Convert a path to an absolute path by adding the current directory if necessary.

Convert a set of paths to an absolute path by joining them together and adding the current directory if necessary. Equivalent to .

– Function.

Normalize a path, removing "." and ".." entries.

– Function.

Canonicalize a path by expanding symbolic links and removing "." and ".." entries.

– Function.

Return a relative filepath to either from the current directory or from an optional start directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of or .

– Function.

On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

– Function.

Split a path into a tuple of the directory name and file name.

– Function.

On Windows, split a path into the drive letter part and the path part. On Unix systems, the first component is always the empty string.

– Function.

If the last component of a path contains a dot, split the path into everything before the dot and everything including and after the dot. Otherwise, return a tuple of the argument unmodified and the empty string.

Chapter 56

I/O and Network

56.1 General I/O

– Constant.

Global variable referring to the standard out stream.

– Constant.

Global variable referring to the standard error stream.

– Constant.

Global variable referring to the standard input stream.

– Function.

Open a file in a mode specified by five boolean arguments. The default is to open files for reading only. Returns a stream for accessing the file.

Alternate syntax for open, where a string-based mode specifier is used instead of the five booleans. The values of correspond to those from or Perl , and are equivalent to setting the following boolean groups:

Mode	Description
r	read
r+	read, write
w	write, create, truncate
w+	read, write, create, truncate
a	write, create, append
a+	read, write, create, append

Apply the function to the result of and close the resulting file descriptor upon completion.

Examples

Start running asynchronously, and return a tuple . If is , then reads from the process's standard output and optionally specifies the process's standard input stream. If is , then writes to the process's standard input and optionally specifies the process's standard output stream.

Similar to , but calls on the resulting process stream, then closes the input stream and waits for the process to complete. Returns the value returned by .

– Type.

Create an , which may optionally operate on a pre-existing array. If the readable/writable arguments are given, they restrict whether or not the buffer may be read from or written to respectively. By default the buffer is readable but not writable. The last argument optionally specifies a size beyond which the buffer may not be grown.

Create an in-memory I/O stream.

Create a fixed size IOBuffer. The buffer will not grow dynamically.

Create a read-only on the data underlying the given string.

Examples

– Method.

Obtain the contents of an as an array, without copying. Afterwards, the is reset to its initial state.

– Function.

Create an object from an integer file descriptor. If is , closing this object will close the underlying descriptor. By default, an is closed when it is garbage collected. allows you to associate the descriptor with a named file.

– Function.

Commit all currently buffered writes to the given stream.

– Function.

Close an I/O stream. Performs a first.

– Method.

Read up to bytes from and return the CRC-32c checksum, optionally mixed with a starting integer. If is not supplied, then will be read until the end of the stream.

– Function.

Write the canonical binary representation of a value to the given I/O stream or file. Returns the number of bytes written into the stream.

You can write multiple values with the same call. i.e. the following are equivalent:

– Function.

Open a file and read its contents. is passed to : this is equivalent to .

Read the entire contents of a file as a string.

Read at most bytes from , returning a of the bytes read.

Read at most bytes from , returning a of the bytes read.

If is (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If is , at most one call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the option.

Read a single value of type from , in canonical binary representation.

Read the entirety of , as a String.

– Function.

Read binary data from an I/O stream or file, filling in .

– Function.

Read at most bytes from into , returning the number of bytes read. The size of will be increased if needed (i.e. if is greater than and enough bytes could be read), but it will never be decreased.

Read at most bytes from into , returning the number of bytes read. The size of will be increased if needed (i.e. if is greater than and enough bytes could be read), but it will never be decreased.

See for a description of the option.

– Function.

Copy from the stream object into (converted to a pointer).

It is recommended that subtypes override the following method signature to provide more efficient implementations:

– Function.

Copy from (converted to a pointer) into the object.

It is recommended that subtypes override the following method signature to provide more efficient implementations:

– Function.

Get the current position of a stream.

– Function.

Seek a stream to the given position.

– Function.

Seek a stream to its beginning.

– Function.

Seek a stream to its end.

– Function.

Seek a stream relative to the current position.

– Function.

Add a mark at the current position of stream . Returns the marked position.

See also , , .

– Function.

Remove a mark from stream . Returns if the stream was marked, otherwise.

See also , , .

– Function.

Reset a stream to a previously marked position, and remove the mark. Returns the previously marked position. Throws an error if the stream is not marked.

See also , , .

– Function.

Returns if stream is marked.

See also , , .

– Function.

Tests whether an I/O stream is at end-of-file. If the stream is not yet exhausted, this function will block to wait for more data if necessary, and then return . Therefore it is always safe to read one byte after seeing return . will return as long as buffered data is still available, even if the remote end of a connection is closed.

– Function.

Determine whether a stream is read-only.

– Function.

Returns if the specified IO object is writable (if that can be determined).

– Function.

Returns if the specified IO object is readable (if that can be determined).

– Function.

Determine whether an object - such as a stream, timer, or mmap – is not yet closed. Once an object is closed, it will never produce a new event. However, a closed stream may still have data to read in its buffer, use to check for the ability to read data. Use to be notified when a stream might be writable or readable.

– Function.

Write an arbitrary value to a stream in an opaque format, such that it can be read back by . The read-back value will be as identical as possible to the original. In general, this process will not work if the reading and writing are done by different versions of Julia, or an instance of Julia with a different system image. values are serialized as all-zero bit patterns ().

– Function.

Read a value written by . assumes the binary data read from is correct and has been serialized by a compatible implementation of . It has been designed with simplicity and performance as a goal and does not validate the data read. Malformed data can result in process termination. The caller has to ensure the integrity and correctness of data read from .

– Function.

Print the shortest possible representation, with the minimum number of consecutive non-zero digits, of number , ensuring that it would parse to the exact same number.

– Function.

Returns the file descriptor backing the stream or file. Note that this function only applies to synchronous 's and 's not to any of the asynchronous streams.

– Function.

Create a pipe to which all C and Julia level output will be redirected. Returns a tuple representing the pipe ends. Data written to may now be read from the end of the pipe. The end is given for convenience in case the old object was cached by the user and needs to be replaced elsewhere.

Note
must be a , a , or a .

– Method.

Run the function while redirecting to . Upon completion, is restored to its prior setting.

Note
must be a , a , or a .

– Function.

Like , but for .

Note
must be a , a , or a .

– Method.

Run the function while redirecting to . Upon completion, is restored to its prior setting.

Note
must be a , a , or a .

– Function.

Like , but for . Note that the order of the return tuple is still , i.e. data to be read from may be written to .

Note
must be a , a , or a .

– Method.

Run the function while redirecting to . Upon completion, is restored to its prior setting.

Note
must be a , a , or a .

– Function.

Read the entirety of as a string and remove a single trailing newline. Equivalent to .

– Function.

Resize the file or buffer given by the first argument to exactly bytes, filling previously unallocated space with '\0' if the file or buffer is grown.

– Function.

Advance the stream until before the first character for which returns . For example will skip all whitespace. If keyword argument is specified, characters from that character through the end of a line will also be skipped.

– Function.

Read until the end of the stream/file and count the number of lines. To specify a file pass the filename as the first argument. EOL markers other than are supported by passing them as the second argument.

– Function.

An that allows reading and performs writes by appending. Seeking and truncating are not supported. See for the available constructors. If is given, creates a to operate on a data vector, optionally specifying a size beyond which the underlying may not be grown.

– Function.

Read all available data on the stream, blocking the task only if no data is available. The result is a .

– Type.

provides a mechanism for passing output configuration settings among methods.

In short, it is an immutable dictionary that is a subclass of . It supports standard dictionary operations such as , and can also be used as an I/O stream.

– Method.

Create an that wraps a given stream, adding the specified pair to the properties of that stream (note that can itself be an).

- use to see if this particular combination is in the properties set
- use to retrieve the most recent value for a particular key

The following properties are in common use:

- : Boolean specifying that small values should be printed more compactly, e.g. that numbers should be printed with fewer digits. This is set when printing array elements.
- : Boolean specifying that containers should be truncated, e.g. showing in place of most elements.
- : A giving the size in rows and columns to use for text output. This can be used to override the display size for called functions, but to get the size of the screen use the function.

– Method.

Create an that wraps an alternate but inherits the properties of .

56.2 Text I/O

– Method.

Write an informative text representation of a value to the current output stream. New types should overload where the first argument is a stream. The representation used by generally includes Julia-specific formatting and type information.

– Function.

Show a compact representation of a value to . If is not specified, the default is to print to .

This is used for printing array elements without repeating type information (which would be redundant with that printed once for the whole array), and without line breaks inside the representation of an element.

To offer a compact representation different from its standard one, a custom type should test in its normal method.

– Function.

Return a string giving a brief description of a value. By default returns , e.g. .

For arrays, returns a string of size and type info, e.g. .

– Function.

Write to (or to the default output stream if is not given) a canonical (un-decorated) text representation of a value if there is one, otherwise call . The representation used by includes minimal formatting and tries to avoid Julia-specific details.

Examples

```
```

– Function.

```
```

Print (using) followed by a newline. If is not supplied, prints to .

– Function.

```
```

Print in a color specified as a symbol.

may take any of the values , or or an integer between 0 and 255 inclusive. Note that not all terminals support 256 colors. If the keyword is given as , the result will be printed in bold.

– Function.

```
```

Display an informational message. Argument is a string describing the information to be displayed. The keyword argument can be used to override the default prepending of .

Examples

```
```

See also .

– Function.

```
```

Display a warning. Argument is a string describing the warning to be displayed. Set to true and specify a to only display the first time is called. If is not a backtrace is displayed. If is not both it and are displayed.

See also .

Display a warning. Argument is a string describing the warning to be displayed.

Examples

– Function.

Stream output of informational, warning, and/or error messages to , overriding what was otherwise specified. Optionally, divert stream only for module , or specifically function within . can be (the default), , , or . See for the current set of redirections. Call with no arguments (or just the) to reset everything.

– Macro.

Print using C style format specification string, with some caveats: and are printed consistently as and for flags , , , , , , and . Furthermore, if a floating point number is equally close to the numeric values of two possible output strings, the output string further away from zero is chosen.

Optionally, an may be passed as the first argument to redirect output.

Examples

– Macro.

Return formatted output as string.

Examples

– Function.

Call the given function with an I/O stream and the supplied extra arguments. Everything written to this I/O stream is returned as a string.

Examples

– Function.

Show a descriptive representation of an exception object.

– Function.

Show every part of the representation of a value.

– Macro.

Show every part of the representation of the given expression. Equivalent to .

– Function.

Read a single line of text from the given I/O stream or file (defaults to). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with or or the end of an input stream. When is true (as it is by default), these trailing newline characters are removed from the line before it is returned. When is false, they are returned as part of the line.

– Function.

Read a string from an I/O stream or a file, up to and including the given delimiter byte. The text is assumed to be encoded in UTF-8.

– Function.

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading repeatedly with the same arguments and saving the resulting lines as a vector of strings.

– Function.

Create an iterable object that will yield each line from an I/O stream or a file. Iteration calls on the stream argument repeatedly with passed through, determining whether trailing end-of-line characters are removed. When called with a file name, the file is opened once at the beginning of iteration and closed at the end. If iteration is interrupted, the file will be closed when the object is garbage collected.

– Method.

Read a matrix from the source where each line (separated by) gives one row, with elements separated by the given delimiter. The source can be a text file, stream or byte array. Memory mapped files can be used by passing the byte array representation of the mapped segment as source.

If is a numeric type, the result is an array of that type, with any non-numeric elements as for floating-point types, or zero. Other useful values of include , , and .

If is , the first row of data will be read as header and the tuple is returned instead of only .

Specifying will ignore the corresponding number of initial lines from the input.

If is , blank lines in the input will be ignored.

If is , the file specified by is memory mapped for potential speedups. Default is except on Windows. On Windows, you may want to specify if the file is large, and is only read once and not written to.

If is , columns enclosed within double-quote (") characters are allowed to contain new lines and column delimiters. Double-quote characters within a quoted field must be escaped with another double-quote. Specifying as a tuple of the expected rows and columns (including header, if any) may speed up reading of large files. If is , lines beginning with and text following in any line are ignored.

– Method.

If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

– Method.

The end of line delimiter is taken as .

– Method.

The end of line delimiter is taken as . If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

– Method.

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as .

– Method.

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as . If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

– Function.

Write (a vector, matrix, or an iterable collection of iterable rows) as text to (either a filename string or an stream) using the given delimiter (which defaults to tab, but can be any printable Julia object, typically a or).

For example, two vectors and of the same length can be written as two columns of tab-delimited text to by either or by .

– Function.

Equivalent to with set to comma, and type optionally defined by .

– Function.

Equivalent to with set to comma.

– Type.

Returns a new write-only I/O stream, which converts any bytes written to it into base64-encoded ASCII bytes written to . Calling on the stream is necessary to complete the encoding (but does not close).

Examples

– Type.

Returns a new read-only I/O stream, which decodes base64-encoded data read from .

Examples

– Function.

Given a -like function , which takes an I/O stream as its first argument, calls to write to a base64-encoded string, and returns the string. is equivalent to : it converts its arguments into bytes using the standard functions and returns the base64-encoded string.

See also .

– Function.

Decodes the base64-encoded and returns a of the decoded bytes.

See also

Examples

– Function.

Return the nominal size of the screen that may be used for rendering output to this io object

56.3 Multimedia I/O

Just as text output is performed by and user-defined types can indicate their textual representation by overloading , Julia provides a standardized mechanism for rich multimedia output (such as images, formatted text, or even audio and video), consisting of three parts:

- A function to request the richest available multimedia display of a Julia object (with a plain-text fallback).
- Overloading allows one to indicate arbitrary multimedia representations (keyed by standard MIME types) of user-defined types.
- Multimedia-capable display backends may be registered by subclassing a generic type and pushing them onto a stack of display backends via .

The base Julia runtime provides only plain-text display, but richer displays may be enabled by loading external modules or by using graphical Julia environments (such as the IPython-based IJulia notebook).

– Function.

Display using the topmost applicable display in the display stack, typically using the richest supported multimedia output for , with plain-text output as a fallback. The variant attempts to display on the given display only, throwing a if cannot display objects of this type.

There are also two variants with a argument (a MIME type string, such as), which attempt to display using the requested MIME type *only*, throwing a if this type is not supported by either the display(s) or by . With these variants, one can also supply the "raw" data in the requested MIME type by passing (for MIME types with text-based storage, such as text/html or application/postscript) or (for binary MIME types).

– Function.

By default, the functions simply call . However, some display backends may override to modify an existing display of (if any). Using is also a hint to the backend that may be redisplayed several times, and the backend may choose to defer the display until (for example) the next interactive prompt.

– Function.

Returns a boolean value indicating whether the given type (string) is displayable by any of the displays in the current display stack, or specifically by the display in the second variant.

– Method.

The functions ultimately call in order to write an object as a given type to a given I/O (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type , it is only necessary to define a new method for , via: , where is a MIME-type string and the function body calls (or similar) to write that representation of to . (Note that the notation only supports literal strings; to construct types in a more flexible manner use .)

For example, if you define a type and know how to write it to a PNG file, you could define a function to allow your images to be displayed on any PNG-capable (such as IJulia). As usual, be sure to in order to add new methods to the built-in Julia function .

The default MIME type is . There is a fallback definition for output that calls with 2 arguments. Therefore, this case should be handled by defining a 2-argument method.

Technically, the macro defines a singleton type for the given string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The first argument to can be an specifying output format properties. See for details.

– Function.

Returns a boolean value indicating whether or not the object can be written as the given type. (By default, this is determined automatically by the existence of the corresponding method for .)

Examples

– Function.

Returns an or containing the representation of in the requested type, as written by (throwing a if no appropriate is available). An is returned for MIME types with textual representations (such as or), whereas binary data is returned as . (The function returns whether or not Julia treats a given type as text.)

As a special case, if is an (for textual MIME types) or a (for binary MIME types), the function assumes that is already in the requested format and simply returns . This special case does not apply to the MIME type. This is useful so that raw data can be passed to .

– Function.

Returns an containing the representation of in the requested type. This is similar to except that binary data is base64-encoded as an ASCII string.

As mentioned above, one can also define new display backends. For example, a module that can display PNG images in a window can register this capability with Julia, so that calling on types with PNG representations will automatically display the image using the module's window.

In order to define a new display backend, one should first create a subtype of the abstract class . Then, for each MIME type (string) that can be displayed on , one should define a function that displays as that MIME type, usually by calling . A should be thrown if cannot be displayed as that MIME type; this is automatic if one calls . Finally, one should define a function that queries for the types supported by and displays the "best" one; a should be thrown if no supported MIME types are found for . Similarly, some subtypes may wish to override . (Again, one should to add new methods to .) The return values of these functions are up to the implementation (since in some cases it may be useful to return a display "handle" of some type). The display functions for can then be called directly, but they can also be invoked automatically from simply by pushing a new display onto the display-backend stack with:

– Function.

Pushes a new display on top of the global display-backend stack. Calling or will display on the topmost compatible backend in the stack (i.e., the topmost backend that does not throw a).

– Function.

Pop the topmost backend off of the display-backend stack, or the topmost copy of in the second variant.

– Type.

Returns a , which displays any object as the text/plain MIME type (by default), writing the text representation to the given I/O stream. (This is how objects are printed in the Julia REPL.)

– Function.

Determine whether a MIME type is text data. MIME types are assumed to be binary data except for a set of types known to be text data (possibly Unicode).

Examples

56.4 Memory-mapped I/O

– Type.

Create an -like object for creating zeroed-out mmapped-memory that is not tied to a file for use in . Used by for creating shared memory arrays.

– Method.

Create an whose values are linked to a file, using memory-mapping. This provides a convenient way of working with data too large to fit in the computer's memory.

The type is an with a bits-type element of and dimension that determines how the bytes of the array are interpreted. Note that the file must be stored in binary format, and no format conversions are possible (this is a limitation of operating systems, not Julia).

is a tuple or single specifying the size or length of the array.

The file is passed via the stream argument, either as an open or filename string. When you initialize the stream, use for a "read-only" array, and to create a new array used to write values to disk.

If no argument is specified, the default is .

Optionally, you can specify an offset (in bytes) if, for example, you want to skip over a header in the file. The default value for the offset is the current stream position for an .

The keyword argument specifies whether the disk file should be grown to accommodate the requested size of array (if the total file size is < requested array size). Write privileges are required to grow the file.

The keyword argument specifies whether the resulting and changes made to it will be visible to other processes mapping the same file.

For example, the following code

creates a -by- , linked to the file associated with stream .

A more portable file would need to encode the word size – 32 bit or 64 bit – and endianness information in the header. In practice, consider encoding binary data using standard formats like HDF5 (which can be used with memory-mapping).

– Method.

Create a whose values are linked to a file, using memory-mapping; it has the same purpose, works in the same way, and has the same arguments, as , but the byte representation is different.

Example:

This would create a 25-by-30000 , linked to the file associated with stream .

– Function.

Forces synchronization between the in-memory version of a memory-mapped or and the on-disk version.

56.5 Network I/O

– Method.

Connect to the host on port .

– Method.

Connect to the named pipe / UNIX domain socket at .

– Method.

Listen on port on the address specified by . By default this listens on only. To listen on all interfaces pass or as appropriate. determines how many connections can be pending (not having called) before the server will begin to reject them. The default value of is 511.

– Method.

Create and listen on a named pipe / UNIX domain socket.

– Function.

Gets the IP address of the (may have to do a DNS lookup)

– Function.

Get the IP address and port that the given socket is bound to.

– Function.

Get the IP address and port of the remote endpoint that the given socket is connected to. Valid only for connected TCP sockets.

– Type.

Returns an IPv4 object from ip address formatted as an .

– Type.

Returns an IPv6 object from ip address formatted as an .

– Function.

Returns the number of bytes available for reading before a read from this stream or buffer will block.

– Function.

Accepts a connection on the given server and returns a connection to the client. An uninitialized client stream may be provided, in which case it will be used instead of creating a new stream.

– Function.

Create a on any port, using hint as a starting point. Returns a tuple of the actual port that the server was created on and the server itself.

– Function.

Monitor a file descriptor for changes in the read or write availability, and with a timeout given by seconds.

The keyword arguments determine which of read and/or write status should be monitored; at least one of them must be set to .

The returned value is an object with boolean fields , , and , giving the result of the polling.

– Function.

Monitor a file for changes by polling every seconds until a change occurs or seconds have elapsed. The should be a long period; the default is 5.007 seconds.

Returns a pair of objects when a change is detected.

To determine when a file was modified, compare to detect notification of changes. However, using for this operation is preferred, since it is more reliable and efficient, although in some situations it may not be available.

– Function.

Watch file or directory for changes until a change occurs or seconds have elapsed.

The returned value is an object with boolean fields , , and , giving the result of watching the file.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

– Function.

Bind to the given . Note that will listen on all devices.

- The parameter disables dual stack mode. If , only an IPv6 stack is created.
- If , multiple threads or processes can bind to the same address without error if they all set , but only the last to bind will receive any traffic.

Associates the lifetime of with a task. Channel is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on .

The object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed Channel objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

– Function.

Send over to .

– Function.

Read a UDP packet from the specified socket, and return the bytes received. This call blocks.

– Function.

Read a UDP packet from the specified socket, returning a tuple of , where will be either IPv4 or IPv6 as appropriate.

– Function.

Set UDP socket options.

- : loopback for multicast packets (default:).

- : TTL for multicast packets (default:).
- : flag must be set to if socket will be used for broadcast messages, or else the UDP system will return an access error (default:).
- : Time-to-live of packets sent on the socket (default:).

– Function.

Converts the endianness of a value from Network byte order (big-endian) to that used by the Host.

– Function.

Converts the endianness of a value from that used by the Host to Network byte order (big-endian).

– Function.

Converts the endianness of a value from Little-endian to that used by the Host.

– Function.

Converts the endianness of a value from that used by the Host to Little-endian.

– Constant.

The 32-bit byte-order-mark indicates the native byte order of the host machine. Little-endian machines will contain the value . Big-endian machines will contain the value .

Chapter 57

Punctuation

Extended documentation for mathematical symbols & functions is here.

symbol	meaning
	invoke macro ; followed by space-separated expressions
	prefix ”not” operator
	at the end of a function name, indicates that a function modifies its argument(s)
	begin single line comment
	begin multi-line comment (these are nestable)
	end multi-line comment
	string and expression interpolation
	remainder operator
	exponent operator
	bitwise and
	short-circuiting boolean and
	bitwise or
	short-circuiting boolean or
	bitwise xor operator
	multiply, or matrix multiply
	the empty tuple
	bitwise not operator
	backslash operator
	complex transpose operator A
	array indexing
	vertical concatenation
	also vertical concatenation
	with space-separated expressions, horizontal concatenation
	parametric type instantiation
	statement separator
	separate function arguments or tuple components
	3-argument conditional operator (conditional ? if_true : if_false)
	delimit string literals
	delimit character literals
	delimit external process (command) specifications
	splice arguments into a function call or declare a varargs function or type
	access named fields in objects/modules, also prefixes elementwise operator/function calls
	range a, a+1, a+2, ..., b
	range a, a+s, a+2s, ..., b
	index an entire dimension (1:end)
	type annotation, depending on context
	quoted expression
	symbol a
	(reverse of subtype operator)

Chapter 58

Sorting and Related Functions

Julia has an extensive, flexible API for sorting and interacting with already-sorted arrays of values. By default, Julia picks reasonable algorithms and sorts in standard ascending order:

You can easily sort in reverse order as well:

To sort an array in-place, use the "bang" version of the sort function:

Instead of directly sorting an array, you can compute a permutation of the array's indices that puts the array into sorted order:

Arrays can easily be sorted according to an arbitrary transformation of their values:

Or in reverse order by a transformation:

If needed, the sorting algorithm can be chosen:

All the sorting and order related functions rely on a ”less than” relation defining a total order on the values to be manipulated. The function is invoked by default, but the relation can be specified via the keyword.

58.1 Sorting Functions

– Function.

Sort the vector in place. is used by default for numeric arrays while is used for other arrays. You can specify an algorithm to use via the keyword (see Sorting Algorithms for available algorithms). The keyword lets you provide a function that will be applied to each element before comparison; the keyword allows providing a custom ”less than” function; use to reverse the sorting order. These options are independent and can be used together in all possible combinations: if both and are specified, the function is applied to the result of the function; reverses whatever ordering specified via the and keywords.

Examples

– Function.

Variant of that returns a sorted copy of leaving itself unmodified.

Examples

Sort a multidimensional array along the given dimension. See for a description of possible keyword arguments.

Examples

– Function.

Return a permutation vector of indices of that puts it in sorted order. Specify to choose a particular sorting algorithm (see Sorting Algorithms). is used by default, and since it is stable, the resulting permutation will be the lexicographically first one that puts the input array into sorted order – i.e. indices of equal elements appear in ascending order. If you choose a non-stable sorting algorithm such as , a different permutation that puts the array into order may be returned. The order is specified using the same keywords as .

See also .

Examples

– Function.

Like , but accepts a preallocated index vector . If is (the default), is initialized to contain the values .

Examples

– Function.

Sort the rows of matrix lexicographically. See for a description of possible keyword arguments.

Examples

– Function.

Sort the columns of matrix lexicographically. See for a description of possible keyword arguments.

Examples

58.2 Order-Related Functions

– Function.

Test whether a vector is in sorted order. The , and keywords modify what order is considered to be sorted just as they do for .

Examples

– Function.

Return the range of indices of which compare as equal to (using binary search) according to the order specified by the , and keywords, assuming that is already sorted in that order. Return an empty range located at the insertion point if does not contain values equal to .

Examples

– Function.

Return the index of the first value in greater than or equal to , according to the specified order. Return if is greater than all values in . is assumed to be sorted.

Examples

– Function.

Return the index of the last value in less than or equal to , according to the specified order. Return if is less than all values in . is assumed to be sorted.

Examples

– Function.

Partially sort the vector in place, according to the order specified by , and so that the value at index (or range of adjacent values if is a range) occurs at the position where it would appear if the array were fully sorted via a non-stable algorithm. If is a single index, that value is returned; if is a range, an array of values at those indices is returned. Note that does not fully sort the input array.

Examples

– Function.

Variant of which copies before partially sorting it, thereby returning the same thing as but leaving unmodified.

– Function.

Return a partial permutation of the vector , according to the order specified by , and , so that returns the first (or range of adjacent values if is a range) values of a fully sorted version of . If is a single index (Integer), an array of the first indices is returned; if is a range, an array of those indices is returned. Note that the handling of integer values for is different from in that it returns a vector of elements instead of just the th element. Also note that this is equivalent to, but more efficient than, calling .

– Function.

Like , but accepts a preallocated index vector . If is (the default), ix is initialized to contain the values .

58.3 Sorting Algorithms

There are currently four sorting algorithms available in base Julia:

-
-
-
-

is an O(n^2) stable sorting algorithm. It is efficient for very small , and is used internally by .

is an O(n log n) sorting algorithm which is in-place, very fast, but not stable – i.e. elements which are considered equal will not remain in the same order in which they originally appeared in the array to be sorted. is the default algorithm for numeric values, including integers and floats.

is similar to , but the output array is only sorted up to index if is an integer, or in the range of if is an . For example:

is an O(n log n) stable sorting algorithm but is not in-place – it requires a temporary array of half the size of the input array – and is typically not quite as fast as . It is the default algorithm for non-numeric data.

The default sorting algorithms are chosen on the basis that they are fast and stable, or *appear* to be so. For numeric types indeed, is selected as it is faster and indistinguishable in this case from a stable sort (unless the array records its mutations in some way). The stability property comes at a non-negligible cost, so if you don't need it, you may want to explicitly specify your preferred algorithm, e.g. .

The mechanism by which Julia picks default sorting algorithms is implemented via the function. It allows a particular algorithm to be registered as the default in all sorting functions for specific arrays. For example, here are the two default methods from :

As for numeric arrays, choosing a non-stable default algorithm for array types for which the notion of a stable sort is meaningless (i.e. when two values comparing equal can not be distinguished) may make sense.

Chapter 59

Package Manager Functions

All package manager functions are defined in the module. None of the module's functions are exported; to use them, you'll need to prefix each function call with an explicit , e.g. or .

Functions for package development (e.g. , , etc.) have been moved to the PkgDev package. See PkgDev README for the documentation of those functions.

– Function.

Returns the absolute path of the package directory. This defaults to on all platforms (i.e. in UNIX shell syntax). If the environment variable is set, then that path is used in the returned value as . If is a relative path, it is interpreted relative to whatever the current working directory is.

Equivalent to – i.e. it appends path components to the package directory and normalizes the resulting path. In particular, returns the path to the package .

– Function.

Initialize as a package directory. This will be done automatically when the is not set and uses its default value. As part of this process, clones a local METADATA git repository from the site and branch specified by its arguments, which are typically not provided. Explicit (non-default) arguments can be used to support a custom METADATA setup.

– Function.

Determines an optimal, consistent set of package versions to install or upgrade to. The optimal set of package versions is based on the contents of and the state of installed packages in , Packages that are no longer required are moved into .

– Function.

Opens in the editor specified by the or environment variables; when the editor command returns, it runs to determine and install a new optimal set of installed package versions.

– Function.

Add a requirement entry for to and call . If are given, they must be objects and they specify acceptable version intervals for .

– Function.

Remove all requirement entries for from and call .

– Function.

If has a URL registered in , clone it from that URL on the default branch. The package does not need to have any registered versions.

Clone a package directly from the git URL . The package does not need to be registered in . The package repo is cloned by the name if provided; if not provided, is determined automatically from .

– Function.

Set the protocol used to access GitHub-hosted packages. Defaults to 'https', with a blank delegating the choice to the package developer.

– Function.

Returns the names of available packages.

Returns the version numbers available for package .

– Function.

Returns a dictionary mapping installed package names to the installed version number of each package.

If is installed, return the installed version number. If is registered, but not installed, return .

– Function.

Prints out a summary of what packages are installed and what version and state they’re in.

Prints out a summary of what version and state , specifically, is in.

– Function.

Update the metadata repo – kept in – then update any fixed packages that can safely be pulled from their origin; then call to determine a new optimal set of packages versions.

Without arguments, updates all installed packages. When one or more package names are provided as arguments, only those packages and their dependencies are updated.

– Function.

Checkout the repo to the branch . Defaults to checking out the ”master” branch. To go back to using the newest compatible released version, use . Changes are merged (fast-forward only) if the keyword argument , and the latest version is pulled from the upstream repo if .

– Function.

Pin at the current version. To go back to using the newest compatible released version, use

Pin at registered version .

– Function.

Free the package to be managed by the package manager again. It calls to determine optimal package versions after. This is an inverse for both and .

You can also supply an iterable collection of package names, e.g., to free multiple packages at once.

– Function.

Run the build scripts for all installed packages in depth-first recursive order.

Run the build script in for each package in and all of their dependencies in depth-first recursive order. This is called automatically by on all installed or updated packages.

– Function.

Run the tests for all installed packages ensuring that each package's test dependencies are installed for the duration of the test. A package is tested by running its file and test dependencies are specified in . Coverage statistics for the packages may be generated by passing . The default behavior is not to run coverage.

Run the tests for each package in ensuring that each package's test dependencies are installed for the duration of the test. A package is tested by running its file and test dependencies are specified in . Coverage statistics for the packages may be generated by passing . The default behavior is not to run coverage.

– Function.

List the packages that have as a dependency.

Chapter 60

Dates and Time

60.1 Dates and Time Types

– Type.

types represent discrete, human representations of time.

– Type.

A is useful for expressing time periods that are not a fixed multiple of smaller periods. For example, ”a year and a day” is not a fixed number of days, but can be expressed using a . In fact, a is automatically generated by addition of different period types, e.g. produces a result.

– Type.

types represent integer-based, machine representations of time as continuous timelines starting from an epoch.

– Type.

The represents a machine timeline based on UT time (1 day = one revolution of the earth). The is a parameter that indicates the resolution or precision of the instant.

– Type.

types wrap machine instances to provide human representations of the machine instant. , and are subtypes of .

– Type.

wraps a and interprets it according to the proleptic Gregorian calendar.

– Type.

wraps a and interprets it according to the proleptic Gregorian calendar.

– Type.

wraps a and represents a specific moment in a 24-hour day.

60.2 Dates Functions

All Dates functions are defined in the module; note that only the , , and functions are exported; to use all other functions, you'll need to prefix each function call with an explicit , e.g. . Alternatively, you can write to bring all exported functions into to be used without the prefix.

– Method.

Construct a type by parts. Arguments must be convertible to .

– Method.

Construct a type by type parts. Arguments may be in any order. DateTime parts not provided will default to the value of .

– Method.

Create a through the adjuster API. The starting point will be constructed from the provided arguments, and will be adjusted until returns . The step size in adjusting can be provided manually through the keyword. provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that is never satisfied).

Examples

– Method.

Converts a to a . The hour, minute, second, and millisecond parts of the new are assumed to be zero.

– Method.

Construct a by parsing the date time string following the pattern given in the string.

This method creates a object each time it is called. If you are parsing many date time strings of the same format, consider creating a object once and using that as the second argument instead.

– Function.

Format the token from and write it to . The formatting can be based on .

All subtypes of must define this method in order to be able to print a Date / DateTime object according to a containing that token.

– Type.

Construct a date formatting object that can be used for parsing date strings or formatting a date object as a string. The following character codes can be used to construct the string:

Code	Matches	Comment
	1996, 96	Returns year of 1996, 0096
	1996, 96	Returns year of 1996, 0096. Equivalent to
	1, 01	Matches 1 or 2-digit months
	Jan	Matches abbreviated months according to the keyword
	January	Matches full month names according to the keyword
	1, 01	Matches 1 or 2-digit days
	00	Matches hours
	00	Matches minutes
	00	Matches seconds
	.500	Matches milliseconds
	Mon, Tues	Matches abbreviated days of the week
	Monday	Matches full name days of the week
	19960101	Matches fixed-width year, month, and day

Characters not listed above are normally treated as delimiters between date and time slots. For example a string of ”1996-01-15T00:00:00.0” would have a string like ”y-m-dTH:M:S.s”. If you need to use a code character as a delimiter you can escape it using backslash. The date ”1995y01m” would have the format ”y\ym\m”.

Creating a DateFormat object is expensive. Whenever possible, create it once and use it many times or try the string macro. Using this macro creates the DateFormat object once at macro expansion time and reuses it later. see .

See and for how to use a DateFormat object to parse and write Date strings respectively.

– Macro.

Create a object. Similar to but creates the DateFormat object once during macro expansion.

See for details about format specifiers.

– Method.

Construct a by parsing the date time string following the pattern given in the object. Similar to but more efficient when repeatedly parsing similarly formatted date time strings with a pre-created object.

– Method.

Construct a type by parts. Arguments must be convertible to .

– Method.

Construct a type by type parts. Arguments may be in any order. parts not provided will default to the value of .

– Method.

Create a through the adjuster API. The starting point will be constructed from the provided arguments, and will be adjusted until returns . The step size in adjusting can be provided manually through the keyword. provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (given that is never satisfied).

Examples

– Method.

Converts a to a . The hour, minute, second, and millisecond parts of the are truncated, so only the year, month and day parts are used in construction.

– Method.

Construct a by parsing the date string following the pattern given in the string.

This method creates a object each time it is called. If you are parsing many date strings of the same format, consider creating a object once and using that as the second argument instead.

– Method.

Parse a date from a date string using a object .

– Method.

Construct a type by parts. Arguments must be convertible to .

– Method.

Construct a type by type parts. Arguments may be in any order. parts not provided will default to the value of .

– Method.

Create a through the adjuster API. The starting point will be constructed from the provided arguments, and will be adjusted until returns . The step size in adjusting can be provided manually through the keyword. provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that is never satisfied). Note that the default step will adjust to allow for greater precision for the given arguments; i.e. if hour, minute, and second arguments are provided, the default step will be instead of .

Examples

– Method.

Converts a to a . The hour, minute, second, and millisecond parts of the are used to create the new . Microsecond and nanoseconds are zero by default.

– Method.

Returns a corresponding to the user's system time including the system timezone locale.

– Method.

Returns a corresponding to the user's system time as UTC/GMT.

– Function.

Returns for values, for values, and for values.

Accessor Functions

– Function.

The year of a or as an .

– Function.

The month of a or as an .

– Function.

Return the ISO week date of a or as an . Note that the first week of a year is the week that contains the first Thursday of the year, which can result in dates prior to January 4th being in the last week of the previous year. For example, is the 53rd week of 2004.

Examples

– Function.

The day of month of a or as an .

– Function.

The hour of day of a as an .

The hour of a as an .

– Function.

The minute of a as an .

The minute of a as an .

– Function.

The second of a as an .

The second of a as an .

– Function.

The millisecond of a as an .

The millisecond of a as an .

– Function.

The microsecond of a as an .

– Function.

The nanosecond of a as an .

– Method.

Construct a object with the given value. Input must be losslessly convertible to an .

– Method.

Construct a object with the given value. Input must be losslessly convertible to an .

– Method.

Construct a object with the given value. Input must be losslessly convertible to an .

– Method.

Construct a object with the given value. Input must be losslessly convertible to an .

– Method.

The hour part of a DateTime as a .

– Method.

The minute part of a DateTime as a .

– Method.

The second part of a DateTime as a .

– Method.

The millisecond part of a DateTime as a .

– Method.

The microsecond part of a Time as a .

– Method.

The nanosecond part of a Time as a .

– Function.

Simultaneously return the year and month parts of a or .

– Function.

Simultaneously return the month and day parts of a or .

– Function.

Simultaneously return the year, month and day parts of a or .

Query Functions

– Function.

Return the full day name corresponding to the day of the week of the or in the given .

Examples

– Function.

Return the abbreviated name corresponding to the day of the week of the or in the given .

Examples

– Function.

Returns the day of the week as an with .

Examples

– Function.

The day of month of a or as an .

– Function.

For the day of week of , returns which number it is in 's month. So if the day of the week of is Monday, then In the range 1:5.

Examples

"'jldoctest julia> Dates.dayofweekofmonth(Date("2000-02-01")) 1

julia> Dates.dayofweekofmonth(Date("2000-02-08")) 2

julia> Dates.dayofweekofmonth(Date("2000-02-15")) 3 ""

– Function.

For the day of week of , returns the total number of that day of the week in 's month. Returns 4 or 5. Useful in temporal expressions for specifying the last day of a week in a month by including in the adjuster function.

Examples

– Function.

Return the full name of the month of the or in the given .

Examples

– Function.

Return the abbreviated month name of the or in the given .

Examples

– Function.

Returns the number of days in the month of . Value will be 28, 29, 30, or 31.

Examples

– Function.

Returns if the year of is a leap year.

Examples

– Function.

Returns the day of the year for with January 1st being day 1.

– Function.

Returns 366 if the year of is a leap year, otherwise returns 365.

Examples

– Function.

Returns the quarter that resides in. Range of value is 1:4.

– Function.

Returns the day of the current quarter of . Range of value is 1:92.

Adjuster Functions

– Method.

Truncates the value of according to the provided type.

Examples

– Function.

Adjusts to the Monday of its week.

Examples

– Function.

Adjusts to the Sunday of its week.

Examples

– Function.

Adjusts to the first day of its month.

Examples

– Function.

Adjusts to the last day of its month.

Examples

```
```

– Function.

```
```

Adjusts to the first day of its year.

Examples

```
```

– Function.

```
```

Adjusts to the last day of its year.

Examples

```
```

– Function.

```
```

Adjusts to the first day of its quarter.

Examples

```
```

– Function.

```
```

Adjusts to the last day of its quarter.

Examples

– Method.

Adjusts to the next day of week corresponding to with . Setting allows the current to be considered as the next , allowing for no adjustment to occur.

– Method.

Adjusts to the previous day of week corresponding to with . Setting allows the current to be considered as the previous , allowing for no adjustment to occur.

– Function.

Adjusts to the first of its month. Alternatively, will adjust to the first of the year.

– Function.

Adjusts to the last of its month. Alternatively, will adjust to the last of the year.

– Method.

Adjusts by iterating at most iterations by increments until returns . must take a single argument and return a . allows to be considered in satisfying .

– Method.

Adjusts by iterating at most iterations by increments until returns . must take a single argument and return a . allows to be considered in satisfying .

Periods

– Method.

Construct a type with the given value. Input must be losslessly convertible to an .

– Method.

Construct a from a of s. All s of the same type will be added together.

Examples

– Function.

Returns a sensible ”default” value for the input Period by returning for Year, Month, and Day, and for Hour, Minute, Second, and Millisecond.

Rounding Functions

and values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with , , or .

– Method.

Returns the nearest or less than or equal to at resolution .

For convenience, may be a type instead of a value: is a shortcut for .

– Method.

Returns the nearest or greater than or equal to at resolution .

For convenience, may be a type instead of a value: is a shortcut for .

– Method.

Returns the or nearest to at resolution . By default (), ties (e.g., rounding 9:30 to the nearest hour) will be rounded up.

For convenience, may be a type instead of a value: is a shortcut for .

Valid rounding modes for are (default), (), and ().

The following functions are not exported:

– Function.

Simultaneously return the and of a or at resolution . More efficient than calling both and individually.

– Function.

Takes the number of days since the rounding epoch () and returns the corresponding .

– Function.

Takes the number of milliseconds since the rounding epoch () and returns the corresponding .

– Function.

Takes the given and returns the number of days since the rounding epoch () as an .

– Function.

Takes the given and returns the number of milliseconds since the rounding epoch () as an .

Conversion Functions

– Function.

Returns the date portion of .

– Function.

Takes the number of seconds since unix epoch and converts to the corresponding .

– Function.

Takes the given and returns the number of seconds since the unix epoch as a .

– Function.

Takes the number of Julian calendar days since epoch and returns the corresponding .

– Function.

Takes the given and returns the number of Julian calendar days since the julian epoch as a .

– Function.

Takes the number of Rata Die days since epoch and returns the corresponding .

– Function.

Returns the number of Rata Die days since epoch from the given or .

Constants

Days of the Week:

Variable	Abbr.	Value (Int)
		1
		2
		3
		4
		5
		6
		7

Months of the Year:

Variable	Abbr.	Value (Int)
		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
		11
		12

Chapter 61

Iteration utilities

– Function.

For a set of iterable objects, returns an iterable of tuples, where the th tuple contains the th component of each input iterable.

Note that is its own inverse: .

Examples

– Function.

An iterator that yields where is a counter starting at 1, and is the th value from the given iterator. It's useful when you need not only the values over which you are iterating, but also the number of iterations so far. Note that may not be valid for indexing ; it's also possible that , if has indices that do not start at 1. See the method if you want to ensure that is an index.

Examples

An iterator that accesses each element of the array , returning , where is the index for the element and . This is similar to , except will always be a valid index for .

Specifying ensures that will be an integer; specifying ensures that will be a ; specifying chooses whichever has been defined as the native indexing style for array .

Examples

Note that returns as a *counter* (always starting at 1), whereas returns as an *index* (starting at the first linear index of , which may or may not be 1).

See also: , .

– Function.

An iterator that yields the same elements as , but starting at the given .

Examples

```

```

– Function.

```

```

An iterator that counts forever, starting at and incrementing by .

Examples

```

```

– Function.

```

```

An iterator that generates at most the first elements of .

Examples

```

```

– Function.

```

```

An iterator that generates all but the first elements of .

Examples

```

```

– Function.

```

```

An iterator that cycles through forever.

Examples

```

```

– Function.

```

```

An iterator that generates the value forever. If is specified, generates that many times (equivalent to).

Examples

```

```

– Function.

Returns an iterator over the product of several iterators. Each generated element is a tuple whose th element comes from the th argument iterator. The first iterator changes the fastest.

Examples

– Function.

Given an iterator that yields iterators, return an iterator that yields the elements of those iterators. Put differently, the elements of the argument iterator are concatenated.

Examples

– Function.

Iterate over a collection elements at a time.

Examples

Chapter 62

Unit Testing

62.1 Testing Base Julia

Julia is under rapid development and has an extensive test suite to verify functionality across multiple platforms. If you build Julia from source, you can run this test suite with . In a binary install, you can run the test suite using .

– Function.

Run the Julia unit tests listed in , which can be either a string or an array of strings, using processors. (not exported)

62.2 Basic Unit Tests

The module provides simple *unit testing* functionality. Unit testing is a way to see if your code is correct by checking that the results are what you expect. It can be helpful to ensure your code still works after you make changes, and can be used when developing as a way of specifying the behaviors your code should have when complete.

Simple unit testing can be performed with the and macros:

– Macro.

Tests that the expression evaluates to . Returns a if it does, a if it is , and an if it could not be evaluated.

The form is equivalent to writing which can be useful when the expression is a call using infix syntax such as approximate comparisons:

This is equivalent to the uglier test . It is an error to supply more than one expression unless the first is a call expression and the rest are assignments ().

– Macro.

Tests that the expression throws . The exception may specify either a type, or a value (which will be tested for equality by comparing fields). Note that does not support a trailing keyword form.

For example, suppose we want to check our new function works as expected:

If the condition is true, a is returned:

If the condition is false, then a is returned and an exception is thrown:

If the condition could not be evaluated because an exception was thrown, which occurs in this case because is not defined for symbols, an object is returned and an exception is thrown:

If we expect that evaluating an expression *should* throw an exception, then we can use to check that this occurs:

62.3 Working with Test Sets

Typically a large number of tests are used to make sure functions work correctly over a range of inputs. In the event a test fails, the default behavior is to throw an exception immediately. However, it is normally preferable to run the rest of the tests first to get a better picture of how many errors there are in the code being tested.

The macro can be used to group tests into *sets*. All the tests in a test set will be run, and at the end of the test set a summary will be printed. If any of the tests failed, or could not be evaluated due to an error, the test set will then throw a .

– Macro.

Starts a new test set, or multiple test sets if a loop is provided.

If no custom testset type is given it defaults to creating a . records all the results and, if there are any s or s, throws an exception at the end of the top-level (non-nested) test set, along with a summary of the test results.

Any custom testset type (subtype of) can be given and it will also be used for any nested invocations. The given options are only applied to the test set where they are given. The default test set type does not take any options.

The description string accepts interpolation from the loop indices. If no description is provided, one is constructed based on the variables.

By default the macro will return the testset object itself, though this behavior can be customized in other testset types. If a loop is used then the macro collects and returns a list of the return values of the method, which by default will return a list of the testset objects used in each iteration.

We can put our tests for the function in a test set:

Test sets can also be nested:

In the event that a nested test set has no failures, as happened here, it will be hidden in the summary. If we do have a test failure, only the details for the failed test sets will be shown:

62.4 Other Test Macros

As calculations on floating-point values can be imprecise, you can perform approximate equality checks using either (where , typed via tab completion of , is the function) or use directly.

– Macro.

Tests that the call expression returns a value of the same type inferred by the compiler. It is useful to check for type stability.

can be any call expression. Returns the result of if the types match, and an if it finds different types.

– Macro.

Test whether evaluating results in output that contains the string or matches the regular expression. If is a boolean function, tests whether returns . If is a tuple or array, checks that the error output contains/matches each item in . Returns the result of evaluating .

See also to check for the absence of error output.

– Macro.

Test whether evaluating results in empty output (no warnings or other messages). Returns the result of evaluating .

62.5 Broken Tests

If a test fails consistently it can be changed to use the macro. This will denote the test as if the test continues to fail and alerts the user via an if the test succeeds.

– Macro.

Indicates a test that should pass but currently consistently fails. Tests that the expression evaluates to or causes an exception. Returns a if it does, or an if the expression evaluates to .

The form works as for the macro.

is also available to skip a test without evaluation, but counting the skipped test in the test set reporting. The test will not run but gives a .

– Macro.

Marks a test that should not be executed but should be included in test summary reporting as . This can be useful for tests that intermittently fail, or tests of not-yet-implemented functionality.

The form works as for the macro.

62.6 Creating Custom Types

Packages can create their own subtypes by implementing the and methods. The subtype should have a one-argument constructor taking a description string, with any options passed in as keyword arguments.

– Function.

Record a result to a testset. This function is called by the infrastructure each time a contained macro completes, and is given the test result (which could be an). This will also be called with an if an exception is thrown inside the test block but outside of a context.

– Function.

Do any final processing necessary for the given testset. This is called by the infrastructure after a test block executes. One common use for this function is to record the testset to the parent's results list, using .

takes responsibility for maintaining a stack of nested testsets as they are executed, but any result accumulation is the responsibility of the subtype. You can access this stack with the and methods. Note that these functions are not exported.

– Function.

Retrieve the active test set from the task's local storage. If no test set is active, use the fallback default test set.

– Function.

Returns the number of active test sets, not including the defaut test set

also makes sure that nested invocations use the same subtype as their parent unless it is set explicitly. It does not propagate any properties of the testset. Option inheritance behavior can be implemented by packages using the stack infrastructure that provides.

Defining a basic subtype might look like:

And using that testset looks like:

Chapter 63

C Interface

– Keyword.

Call a function in a C-exported shared library, specified by the tuple , where each component is either a string or symbol. Alternatively, may also be used to call a function pointer , such as one returned by .

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each to the will be converted to the corresponding , by automatic insertion of calls to . (See also the documentation for each of these functions for further details.) In most cases, this simply results in a call to .

– Function.

Obtain a pointer to a global variable in a C-exported shared library, specified exactly as in . Returns a , defaulting to if no argument is supplied. The values can be read or written by or , respectively.

– Function.

Generate C-callable function pointer from Julia function. Type annotation of the return value in the callback function is a must for situations where Julia cannot infer the return type automatically.

Examples

– Function.

Convert to a C argument of type where the input must be the return value of .

In cases where would need to take a Julia object and turn it into a , this function should be used to define and perform that conversion.

Be careful to ensure that a Julia reference to exists as long as the result of this function will be used. Accordingly, the argument to this function should never be an expression, only a variable name or field reference. For example, is acceptable, but is not.

The prefix on this function indicates that using the result of this function after the argument to this function is no longer accessible to the program may cause undefined behavior, including program corruption or segfaults, at any later time.

See also

– Function.

Convert to a value to be passed to C code as type , typically by calling .

In cases where cannot be safely converted to , unlike , may return an object of a type different from , which however is suitable for to handle. The result of this function should be kept valid (for the GC) until the result of is not needed anymore. This can be used to allocate memory that will be accessed by the . If multiple objects need to be allocated, a tuple of the objects can be used as return value.

Neither nor should take a Julia object and turn it into a .

– Function.

Load a value of type from the address of the th element (1-indexed) starting at . This is equivalent to the C expression .

The prefix on this function indicates that no validation is performed on the pointer to ensure that it is valid. Incorrect usage may segfault your program or return garbage answers, in the same manner as C.

– Function.

Store a value of type to the address of the th element (1-indexed) starting at . This is equivalent to the C expression .

The prefix on this function indicates that no validation is performed on the pointer to ensure that it is valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

– Method.

Copy elements from a source pointer to a destination, with no checking. The size of an element is determined by the type of the pointers.

The prefix on this function indicates that no validation is performed on the pointers and to ensure that they are valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

– Method.

Copy elements from a source array to a destination, starting at offset in the source and in the destination (1-indexed).

The prefix on this function indicates that no validation is performed to ensure that N is inbounds on either array. Incorrect usage may corrupt or segfault your program, in the same manner as C.

– Method.

Copy all elements from collection to array .

– Method.

Copy elements from collection starting at offset , to array starting at offset . Returns .

– Function.

Get the native address of an array or string element. Be careful to ensure that a Julia reference to exists as long as this pointer will be used. This function is ”unsafe” like .

Calling is generally preferable to this function.

– Method.

Wrap a Julia object around the data at the address given by , without making a copy. The pointer element type determines the array element type. is either an integer (for a 1d array) or a tuple of the array dimensions. optionally specifies whether Julia should take ownership of the memory, calling on the pointer when the array is no longer referenced.

This function is labelled ”unsafe” because it will crash if is not a valid memory address to data of the requested length.

– Function.

Get the memory address of a Julia object as a . The existence of the resulting will not protect the object from garbage collection, so you must ensure that the object remains referenced for the whole time that the will be used.

– Function.

Convert a to an object reference. Assumes the pointer refers to a valid heap-allocated Julia object. If this is not the case, undefined behavior results, hence this function is considered ”unsafe” and should be used with care.

– Function.

Disable Ctrl-C handler during execution of a function on the current task, for calling external code that may call julia code that is not interrupt safe. Intended to be called using block syntax as follows:

This is not needed on worker threads () since the will only be delivered to the master thread. External functions that do not call julia code or julia runtime automatically disable sigint during their execution.

– Function.

Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of .

– Function.

Raises a for with the descriptive string if is

– Type.

A memory address referring to data of type . However, there is no guarantee that the memory is actually valid, or that it actually represents data of the specified type.

– Type.

An object that safely references data of type . This type is guaranteed to point to valid, Julia-allocated memory of the correct type. The underlying data is protected from freeing by the garbage collector as long as the itself is referenced.

When passed as a argument (either as a or type), a object will be converted to a native pointer to the data it references.

There is no invalid (NULL) .

– Type.

Equivalent to the native c-type.

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type.

– Type.

Equivalent to the native c-type.

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type.

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

– Type.

Equivalent to the native c-type ().

Chapter 64

LLVM Interface

– Function.

Call LLVM IR string in the first argument. Similar to an LLVM function block, arguments are available as consecutive unnamed SSA variables (%0, %1, etc.).

The optional declarations string contains external functions declarations that are necessary for llvm to compile the IR string. Multiple declarations can be passed in by separating them with line breaks.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each to will be converted to the corresponding , by automatic insertion of calls to . (see also the documentation for each of these functions for further details). In most cases, this simply results in a call to .

See for usage examples.

Chapter 65

C Standard Library

– Function.

Call from the C standard library.

– Function.

Call from the C standard library.

– Function.

Call from the C standard library.

See warning in the documentation for regarding only using this on memory originally obtained from .

– Function.

Call from the C standard library. Only use this on memory obtained from , not on pointers retrieved from other C libraries. objects obtained from C libraries should be freed by the free functions defined in that library, to avoid assertion failures if multiple libraries exist on the system.

– Function.

Get the value of the C library's . If an argument is specified, it is used to set the value of .

The value of is only valid immediately after a to a C library routine that sets it. Specifically, you cannot call at the next prompt in a REPL, because lots of code is executed between prompts.

– Function.

Convert a system call error code to a descriptive string

– Function.

Call the Win32 function [only available on Windows].

– Function.

Convert a Win32 system call error code to a descriptive string [only available on Windows].

– Method.

Converts a struct to a number of seconds since the epoch.

– Function.

Convert time, given as a number of seconds since the epoch or a , to a formatted string using the given format. Supported formats are the same as those in the standard C library.

– Function.

Parse a formatted time string into a giving the seconds, minute, hour, date, etc. Supported formats are the same as those in the standard C library. On some platforms, timezones will not be parsed correctly. If the result of this function will be passed to to convert it to seconds since the epoch, the field should be filled in manually. Setting it to will tell the C library to use the current system settings to determine the timezone.

– Type.

Convert a number of seconds since the epoch to broken-down format, with fields , , , , , , , , and .

– Function.

Flushes the C and streams (which may have been written to by external C code).

Chapter 66

Dynamic Linker

The names in are not exported and need to be called e.g. as .

– Function.

Load a shared library, returning an opaque handle.

The extension given by the constant (, , or) can be omitted from the string, as it is automatically appended if needed. If is not an absolute path name, then the paths in the array are searched for , followed by the system load path.

The optional flags argument is a bitwise-or of zero or more of , , , , , , and . These are converted to the corresponding flags of the POSIX (and/or GNU libc and/or MacOS) dlopen command, if possible, or are ignored if the specified functionality is not available on the current platform. The default flags are platform specific. On MacOS the default flags are while on other platforms the defaults are . An important usage of these flags is to specify non default behavior for when the dynamic library loader binds library references to exported symbols and if the bound references are put into process local or global scope. For instance allows the library's symbols to be available for usage in other shared libraries, addressing situations where there are dependencies between shared libraries.

– Function.

Similar to , except returns a pointer instead of raising errors.

– Constant.

Enum constant for . See your platform man page for details, if applicable.

– Function.

Look up a symbol from a shared library handle, return callable function pointer on success.

– Function.

Look up a symbol from a shared library handle, silently return pointer on lookup failure.

– Function.

Close shared library referenced by handle.

– Constant.

File extension for dynamic libraries (e.g. dll, dylib, so) on the current platform.

– Function.

Searches for the first library in in the paths in the list, , or system library paths (in that order) which can successfully be dlopen'd. On success, the return value will be one of the names (potentially prefixed by one of the paths in locations). This string can be assigned to a and used as the library name in future 's. On failure, it returns the empty string.

– Constant.

When calling , the paths in this list will be searched first, in order, before searching the system locations for a valid library handle.

Chapter 67

Profiling

– Macro.

runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

The methods in are not exported and need to be called e.g. as .

– Function.

Clear any existing backtraces from the internal buffer.

– Function.

Prints profiling results to (by default,). If you do not supply a vector, the internal buffer of accumulated backtraces will be used.

The keyword arguments can be any combination of:

- – Determines whether backtraces are printed with (default,) or without () indentation indicating tree structure.
- – If , backtraces from C and Fortran code are shown (normally they are excluded).
- – If (default), instruction pointers are merged that correspond to the same line of code.
- – Limits the depth higher than in the format.
- – Controls the order in format. (default) sorts by the source line, whereas sorts in order of number of collected samples.
- – Limits frames that exceed the heuristic noise floor of the sample (only applies to format). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which , where is the number of samples on this line, and is the number of samples for the callee.
- – Limits the printout to only those lines with at least occurrences.

Prints profiling results to . This variant is used to examine results exported by a previous call to . Supply the vector of backtraces and a dictionary of line information.

See for an explanation of the valid keyword arguments.

– Function.

Configure the between backtraces (measured in seconds), and the number of instruction pointers that may be stored. Each instruction pointer corresponds to a single line of code; backtraces generally consist of a long list of instruction pointers. Default settings can be obtained by calling this function with no arguments, and each can be set independently using keywords or in the order .

– Function.

Returns a reference to the internal buffer of backtraces. Note that subsequent operations, like , can affect unless you first make a copy. Note that the values in have meaning only on this machine in the current session, because it depends on the exact memory addresses used in JIT-compiling. This function is primarily for internal use; may be a better choice for most users.

– Function.

”Exports” profiling results in a portable format, returning the set of all backtraces () and a dictionary that maps the (session-specific) instruction pointers in to values that store the file name, function name, and line number. This function allows you to save profiling results for future analysis.

– Function.

Given a previous profiling run, determine who called a particular function. Supplying the filename (and optionally, range of line numbers over which the function is defined) allows you to disambiguate an overloaded method. The returned value is a vector containing a count of the number of calls and line information about the caller. One can optionally supply backtrace obtained from ; otherwise, the current internal profile buffer is used.

– Function.

Clears any stored memory allocation data when running julia with . Execute the command(s) you want to test (to force JIT-compilation), then call . Then execute your command(s) again, quit Julia, and examine the resulting files.

Chapter 68

StackTraces

– Type.

Stack information representing execution context, with the following fields:

- The name of the function containing the execution context.
- The MethodInstance containing the execution context (if it could be found).
- The path to the file containing the execution context.
- The line number in the file containing the execution context.
- True if the code is from C.
- True if the code is from an inlined frame.
- Representation of the pointer to the execution context as returned by .

– Type.

An alias for provided for convenience; returned by calls to and .

– Function.

Returns a stack trace in the form of a vector of s. (By default stacktrace doesn't return C functions, but this can be enabled.) When called without specifying a trace, first calls .

– Function.

Returns the stack trace for the most recent error thrown, rather than the current execution context.

The following methods and types in are not exported and need to be called e.g. as .

– Function.

Given a pointer to an execution context (usually generated by a call to), looks up stack frame context information. Returns an array of frame information for all functions inlined at that point, innermost function first.

– Function.

Takes a (a vector of) and a function name (a) and removes the specified by the function name from the (also removing all frames above the specified function). Primarily used to remove functions from the prior to returning it.

Returns the with all s from the provided removed.

Chapter 69

SIMD Support

Type is intended for building libraries of SIMD operations. Practical use of it requires using . The type is defined as:

It has a special compilation rule: a homogeneous tuple of maps to an LLVM type when is a primitive bits type and the tuple length is in the set {2-6,8-10,16}.

At , the compiler *might* automatically vectorize operations on such tuples. For example, the following program, when compiled with generates two SIMD addition instructions () on x86 systems:

However, since the automatic vectorization cannot be relied upon, future use will mostly be via libraries that use .

Part V

Developer Documentation

Chapter 70

Reflection and introspection

Julia provides a variety of runtime reflection capabilities.

70.1 Module bindings

The exported names for a are available using , which will return an array of elements representing the exported bindings. returns symbols for all bindings in , regardless of export status.

70.2 DataType fields

The names of fields may be interrogated using . For example, given the following type, returns an arrays of elements representing the field names:

The type of each field in a object is stored in the field of the variable itself:

While is annotated as an , was unannotated in the type definition, therefore defaults to the type.

Types are themselves represented as a structure called :

Note that gives the names for each field of itself, and one of these fields is the field observed in the example above.

70.3 Subtypes

The *direct* subtypes of any may be listed using . For example, the abstract has four (concrete) subtypes:

Any abstract subtype will also be included in this list, but further subtypes thereof will not; recursive application of may be used to inspect the full type tree.

70.4 DataType layout

The internal representation of a is critically important when interfacing with C code and several functions are available to inspect these details. returns true if is stored with C-compatible alignment. returns the (byte) offset for field *i* relative to the start of the type.

70.5 Function methods

The methods of any generic function may be listed using . The method dispatch table may be searched for methods accepting a given type using .

70.6 Expansion and lowering

As discussed in the Metaprogramming section, the function gives the unquoted and interpolated expression () form for a given macro. To use , the expression block itself (otherwise, the macro will be evaluated and the result will be passed instead!). For example:

The functions and are used to display S-expr style views and depth-nested detail views for any expression.

Finally, the function gives the form of any expression and is of particular interest for understanding both macros and top-level statements such as function declarations and variable assignments:

70.7 Intermediate and compiled representations

Inspecting the lowered form for functions requires selection of the specific method to display, because generic functions may have many methods with different type signatures. For this purpose, method-specific code-lowering is available using , and the type-inferred form is available using . adds highlighting to the output of (see).

Closer to the machine, the LLVM intermediate representation of a function may be printed using by , and finally the compiled machine code is available using (this will trigger JIT compilation/code generation for any function which has not previously been called).

For convenience, there are macro versions of the above functions which take standard function calls and expand argument types automatically:

(likewise , , , and)

Chapter 71

Documentation of Julia's Internals

71.1 Initialization of the Julia runtime

How does the Julia runtime execute ?

main()

Execution starts at in .

calls to set the C library locale and to initialize the "ios" library (see and Legacy library).

Next is called to process command line options. Note that only deals with options that affect code generation or early initialization. Other options are handled later by in .

stores command line options in the global struct.

julia_init()

in is called by and calls in .

begins by calling again (it does nothing the second time).

is called to zero the signal handler mask.

searches configured paths for the base system image. See Building the Julia system image.

sets up allocation pools and lists for weak refs, preserved values and finalization.

loads and initializes a pre-compiled femtolisp image containing the scanner/parser.

creates type description objects for the built-in types defined in . e.g.

creates the object; initializes the global struct; and sets to the root task.

initializes the LLVM library.

initializes 8-bit serialization tags for builtin values.

If there is no sysimg file () then the and modules are created and is evaluated:

creates the Julia module.

creates a new Julia module containing constant symbols. These define an integer code for each intrinsic function. translates these symbols into LLVM instructions during code generation.

hooks C functions up to Julia function symbols. e.g. the symbol is bound to C function pointer by calling .

creates the global "Main" module and sets .

Note: then sets . is an alias of at this point, so the set by above is overwritten.

calls which repeatedly calls to execute . <!– TODO – drill down into eval? –>

initializes global C pointers to Julia globals defined in .

pre-allocates global boxed integer value objects for values up to 1024. This speeds up allocation of boxed ints later on. e.g.:

iterates over the looking for values and sets the type name's module prefix to .

does "using Base" in the "Main" module.

Note: now reverts to as it was before being set to above.

Platform specific signal handlers are initialized for (OSX, Linux), and (Windows).

Other signals (and) are hooked up to which prints a backtrace.

calls for each deserialized module to run the function.

Finally is hooked up to and calls .

then returns back to in and calls .

sysimg

If there is a sysimg file, it contains a pre-cooked image of the and modules (and whatever else is created by). See Building the Julia system image.

deserializes the saved sysimg into the current Julia runtime environment and initialization continues after below...

Note: (and in general) uses the Legacy library.

true_main()

loads the contents of into .

If a "program" file was supplied on the command line, then calls which calls which repeatedly calls to execute the program.

However, in our example (), looks up and executes it.

Base._start

calls which calls to create an expression object and to execute it.

Base.eval

was mapped to by .

calls , where is the parsed expression .

calls which calls in .

The stack dump below shows how the interpreter works its way through various methods of and before arriving at which does .

calls to write "Hello World!" to . See Libuv wrappers for stdio.:

Stack frame	Source code	Notes
		called though
		function

Since our example has just one function call, which has done its job of printing "Hello World!", the stack now rapidly unwinds back to .

jl_atexit_hook()

calls . This calls for each module, then calls and cleans up libuv handles.

julia_save()

Finally, calls , which if requested on the command line, saves the runtime state to a new system image. See and .

71.2 Julia ASTs

Julia has two representations of code. First there is a surface syntax AST returned by the parser (e.g. the function), and manipulated by macros. It is a structured representation of code as it is written, constructed by from a character stream. Next there is a lowered form, or IR (intermediate representation), which is used by type inference and code generation. In the lowered form there are fewer types of nodes, all macros are expanded, and all control flow is converted to explicit branches and sequences of statements. The lowered form is constructed by .

First we will focus on the lowered form, since it is more important to the compiler. It is also less obvious to the human, since it results from a significant rearrangement of the input syntax.

Lowered form

The following data types exist in lowered form:

- Has a node type indicated by the field, and an field which is a of subexpressions.
- Identifies arguments and local variables by consecutive numbering. is an abstract type with subtypes and . Both types have an integer-valued field giving the slot index. Most slots have the same type at all uses, and so are represented with . The types of these slots are found in the field of their object. Slots that require per-use type annotations are represented with , which has a field.
- Wraps the IR of a method.
- Contains a single number, specifying the line number the next statement came from.
- Branch target, a consecutively-numbered integer starting at 0.
- Unconditional branch.
- Wraps an arbitrary value to reference as data. For example, the function contains a whose field is the symbol , in order to return the symbol itself instead of evaluating it.
- Refers to global variable in module .
- Refers to a consecutively-numbered (starting at 0) static single assignment (SSA) variable inserted by the compiler.
- Marks a point where a variable is created. This has the effect of resetting a variable to undefined.

Expr types

These symbols appear in the field of s in lowered form.

- Function call (dynamic dispatch). is the function to call, are the arguments.
- Function call (static dispatch). is the MethodInstance to call, are the arguments (including the function that is being called, at).
- Reference a static parameter by index.
- Line number and file name metadata. Unlike a , can also contain a file name.
- Conditional branch. If is false, goes to label identified in .
- Assignment.
- Adds a method to a generic function and assigns the result if necessary.

  Has a 1-argument form and a 4-argument form. The 1-argument form arises from the syntax . In the 1-argument form, the argument is a symbol. If this symbol already names a function in the current scope, nothing happens. If the symbol is undefined, a new function is created and assigned to the identifier specified by the symbol. If the symbol is defined but names a non-function, an error is raised. The definition of "names a function" is that the binding is constant, and refers to an object of singleton type. The rationale for this is that an instance of a singleton type uniquely identifies the type to add the method to. When the type has fields, it wouldn't be clear whether the method was being added to the instance or its type.

  The 4-argument form has the following arguments:

  - A function name, or if unknown. If a symbol, then the expression first behaves like the 1-argument form above. This argument is ignored from then on. When this is , it means a method is being added strictly by type, .
  - A of argument type data. is a of the argument types, and is a of type variables corresponding to the method's static parameters.
  - A of the method itself. For "out of scope" method definitions (adding a method to a function that also has methods defined in different scopes) this is an expression that evaluates to a expression.
  - or , identifying whether the method is staged ().
- Declares a (global) variable as constant.

- Has no arguments; simply yields the value .
- Allocates a new struct-like object. First argument is the type. The pseudo-function is lowered to this, and the type is always inserted by the compiler. This is very much an internal-only feature, and does no checking. Evaluating arbitrary expressions can easily segfault.
- Returns its argument as the value of the enclosing function.
- Yields the caught exception inside a block. This is the value of the run time system variable .
- Enters an exception handler (). is the label of the catch block to jump to on error.
- Pop exception handlers. is the number of handlers to pop.
- Controls turning bounds checks on or off. A stack is maintained; if the first argument of this expression is true or false (means bounds checks are disabled), it is pushed onto the stack. If the first argument is , the stack is popped.
- Indicates the beginning or end of a section of code that performs a bounds check. Like , a stack is maintained, and the second argument can be one of: , , or .
- Part of the implementation of quasi-quote. The argument is a surface syntax AST that is simply copied recursively and returned at run time.
- Metadata. is typically a symbol specifying the kind of metadata, and the rest of the arguments are free-form. The following kinds of metadata are commonly used:
  - and : Inlining hints.
  - : enters a sequence of statements from a specified source location.
    - specifies a filename, as a symbol.
    - optionally specifies the name of an (inlined) function that originally contained the code.
  - : returns to the source location before the matching .

Method

A unique'd container describing the shared metadata for a single method.

- , , , ,
  Metadata to uniquely identify the method for the computer and the human.
- Cache of other methods that may be ambiguous with this one.

- Cache of all MethodInstance ever created for this Method, used to ensure uniqueness. Uniqueness is required for efficiency, especially for incremental precompile and tracking of method invalidation.
- The original source code (usually compressed).
- Pointers to non-AST things that have been interpolated into the AST, required by compression of the AST, type-inference, or the generation of native code.
- , , , ,
  Descriptive bit-fields for the source code of this Method.
- /
  The range of world ages for which this method is visible to dispatch.

MethodInstance

A unique'd container describing a single callable signature for a Method. See especially Proper maintenance and care of multi-threading locks for important details on how to modify these fields safely.

- The primary key for this MethodInstance. Uniqueness is guaranteed through a lookup.
- The that this function describes a specialization of. Or a , if this is a top-level Lambda expanded in Module, and which is not part of a Method.
- The values of the static parameters in indexed by . For the at , this is the empty . But for a runtime from the cache, this will always be defined and indexable.
- The inferred return type for the field, which (in most cases) is also the computed return type for the function in general.
- May contain a cache of the inferred source for this function, or other information about the inference result such as a constant return value may be put here (if), or it could be set to to just indicate is inferred.
- The generic jlcall entry point.
- The ABI to use when calling . Some significant ones include:
  - 0 - Not compiled yet
  - 1 - JL_CALLABLE
  - 2 - Constant (value stored in)
  - 3 - With Static-parameters forwarded

  - 4 - Run in interpreter

- /

  The range of world ages for which this method instance is valid to be called.

CodeInfo

A temporary container for holding lowered source code.

-

  An array of statements

-

  An array of symbols giving the name of each slot (argument or local variable).

-

  An array of types for the slots.

-

  A array of slot properties, represented as bit flags:

  - 2 - assigned (only false if there are *no* assignment statements with this var on the left)
  - 8 - const (currently unused for local variables)
  - 16 - statically assigned once
  - 32 - might be used before assigned. This flag is only valid after type inference.

-

  Either an array or an .

  If an , it gives the number of compiler-inserted temporary locations in the function. If an array, specifies a type for each location.

Boolean properties:

-

  Whether this has been produced by type inference.

-

  Whether this should be inlined.

-

  Whether this should should propagate when inlined for the purpose of eliding blocks.

-

  Whether this is known to be a pure function of its arguments, without respect to the state of the method caches or other mutable global state.

Surface syntax AST

Front end ASTs consist entirely of s and atoms (e.g. symbols, numbers). There is generally a different expression head for each visually distinct syntactic form. Examples will be given in s-expression syntax. Each parenthesized list corresponds to an Expr, where the first element is the head. For example corresponds to in Julia.

Input	AST

Calls

syntax:

parses as .

Operators

Most uses of operators are just function calls, so they are parsed with the head . However some operators are special forms (not necessarily function calls), and in those cases the operator itself is the expression head. In julia-parser.scm these are referred to as ”syntactic operators”. Some operators (and) use N-ary parsing; chained calls are parsed as a single N-argument call. Finally, chains of comparisons have their own special expression structure.

Input	AST

Bracketed forms

Macros

Strings

Doc string syntax:

parses as .

Input	AST

Input	AST

Input	AST

Imports and such

Numbers

Julia supports more number types than many scheme implementations, so not all numbers are represented directly as scheme numbers in the AST.

Block forms

A block of statements is parsed as .

If statement:

parses as:

Input	AST

Input	AST

A loop parses as .

A loop parses as . If there is more than one iteration specification, they are parsed as a block: .

and are parsed as 0-argument expressions and .

is parsed as .

A basic function definition is parsed as . A more complex example:

parses as:

Type definition:

parses as:

The first argument is a boolean telling whether the type is mutable.

blocks parse as . If no variable is present after , is . If there is no clause, then the last argument is not present.

71.3 More about types

If you've used Julia for a while, you understand the fundamental role that types play. Here we try to get under the hood, focusing particularly on Parametric Types.

Types and sets (and and /)

It's perhaps easiest to conceive of Julia's type system in terms of sets. While programs manipulate individual values, a type refers to a set of values. This is not the same thing as a collection; for example a of values is itself a single value. Rather, a type describes a set of *possible* values, expressing uncertainty about which value we have.

A *concrete* type describes the set of values whose direct tag, as returned by the function, is . An *abstract* type describes some possibly-larger set of values.

 describes the entire universe of possible values. is a subset of that includes , , and other concrete types. Internally, Julia also makes heavy use of another type known as , which can also be written as . This corresponds to the empty set.

Julia's types support the standard operations of set theory: you can ask whether is a "subset" (subtype) of with . Likewise, you intersect two types using , take their union with , and compute a type that contains their union with :

While these operations may seem abstract, they lie at the heart of Julia. For example, method dispatch is implemented by stepping through the items in a method list until reaching one for which the type of the argument tuple is a subtype of the method signature. For this algorithm to work, it's important that methods be sorted by their specificity, and that the search begins with the most specific methods. Consequently, Julia also implements a partial order on types; this is achieved by functionality that is similar to , but with differences that will be discussed below.

UnionAll types

Julia's type system can also express an *iterated union* of types: a union of types over all values of some variable. This is needed to describe parametric types where the values of some parameters are not known.

For example, :obj: has two parameters as in . If we did not know the element type, we could write , which is the union of for all values of : .

Such a type is represented by a object, which contains a variable (in this example, of type), and a wrapped type (in this example).

Consider the following methods:

The signature of is a type wrapping a tuple type. All but can be called with ; all but can be called with .

Let's look at these types a little more closely:

This indicates that actually names a type. There is one type for each parameter, nested. The syntax is equivalent to ; internally each is instantiated with a particular variable value, one at a time, outermost-first. This gives a natural meaning to the omission of trailing type parameters; gives a type equivalent to .

A is not itself a type, but rather should be considered part of the structure of a type. Type variables have lower and upper bounds on their values (in the fields and). The symbol is purely cosmetic. Internally, s are compared by address, so they are defined as mutable types to ensure that "different" type variables can be distinguished. However, by convention they should not be mutated.

One can construct s manually:

There are convenience versions that allow you to omit any of these arguments except the symbol.

The syntax is lowered to

so it is seldom necessary to construct a manually (indeed, this is to be avoided).

Free variables

The concept of a *free* type variable is extremely important in the type system. We say that a variable is free in type if does not contain the that introduces variable . For example, the type has no free variables, but the part inside of it does have a free variable, .

A type with free variables is, in some sense, not really a type at all. Consider the type , which refers to all homogeneous arrays of arrays. The inner type , seen by itself, might seem to refer to any kind of array. However, every element of the outer array must have the *same* array type, so cannot refer to just any old array. One could say that effectively "occurs" multiple times, and must have the same value each "time".

For this reason, the function in the C API is very important. Types for which it returns true will not give meaningful answers in subtyping and other type functions.

TypeNames

The following two types are functionally equivalent, yet print differently:

These can be distinguished by examining the field of the type, which is an object of type :

In this case, the relevant field is , which holds a reference to the top-level type used to make new types.

The field of points to itself, but for it points back to the original definition of the type.

What about the other fields? assigns an integer to each type. To examine the field, it's helpful to pick a type that is less heavily used than Array. Let's first create our own type:

(The cache is pre-allocated to have length 8, but only the first two entries are populated.) Consequently, when you instantiate a parametric type, each concrete type gets saved in a type cache. However, instances containing free type variables are not cached.

Tuple types

Tuple types constitute an interesting special case. For dispatch to work on declarations like , the type has to be able to accommodate any tuple. Let's check the parameters:

Unlike other types, tuple types are covariant in their parameters, so this definition permits to match any type of tuple:

However, if a variadic () tuple type has free variables it can describe different kinds of tuples:

Notice that when is free with respect to the type (i.e. its binding type is outside the type), only one value must work over the whole type. Therefore a heterogeneous tuple does not match.

Finally, it's worth noting that is distinct:

What is the "primary" tuple-type?

so is indeed the primary type.

Diagonal types

Consider the type . A method with this signature would look like:

According to the usual interpretation of a type, this ranges over all types, including , so this type should be equivalent to . However, this interpretation causes some practical problems.

First, a value of needs to be available inside the method definition. For a call like , it's not clear what should be. It could be , or perhaps . Intuitively, we expect the declaration to mean . To make sure that invariant holds, we need in this method. That implies the method should only be called for arguments of the exact same type.

It turns out that being able to dispatch on whether two values have the same type is very useful (this is used by the promotion system for example), so we have multiple reasons to want a different interpretation of . To make this work we add the following rule to subtyping: if a variable occurs more than once in covariant position, it is restricted to ranging over only concrete types. ("Covariant position" means that only and types occur between an occurrence of a variable and the type that introduces it.) Such variables are called "diagonal variables" or "concrete variables".

So for example, can be seen as , where ranges over all concrete types. This gives rise to some interesting subtyping results. For example is not a subtype of , because it includes some types like where the two elements have different types. and have the non-trivial intersection . However, *is* a subtype of , because in that case occurs only once and so is not diagonal.

Next consider a signature like the following:

In this case, occurs in invariant position inside . That means whatever type of array is passed unambiguously determines the value of -- we say has an *equality constraint* on it. Therefore in this case the diagonal rule is not really necessary, since the array determines and we can then allow and to be of any subtypes of . So variables that occur in invariant position are never considered diagonal. This choice of behavior is slightly controversial -- some feel this definition should be written as

to clarify whether and need to have the same type. In this version of the signature they would, or we could introduce a third variable for the type of if and can have different types.

The next complication is the interaction of unions and diagonal variables, e.g.

Consider what this declaration means. has type . then can have either the same type , or else be of type . So all of the following calls should match:

These examples are telling us something: when is , there are no extra constraints on . It is as if the method signature had . This means that whether a variable is diagonal is not a static property based on where it appears in a type. Rather, it depends on where a variable appears when the subtyping algorithm *uses* it. When has type , we don't need to use the in , so does not "occur". Indeed, we have the following type equivalence:

Subtyping diagonal variables

The subtyping algorithm for diagonal variables has two components: (1) identifying variable occurrences, and (2) ensuring that diagonal variables range over concrete types only.

The first task is accomplished by keeping counters and (in) for each variable in the environment, tracking the number of invariant and covariant occurrences, respectively. A variable is diagonal when .

The second task is accomplished by imposing a condition on a variable's lower bound. As the subtyping algorithm runs, it narrows the bounds of each variable (raising lower bounds and lowering upper bounds) to keep track of the range of variable values for which the subtype relation would hold. When we are done evaluating the body of a type whose variable is diagonal, we look at the final values of the bounds. Since the variable must be concrete, a contradiction occurs if its lower bound could not be a subtype of a concrete type. For example, an abstract type like cannot be a subtype of a concrete type, but a concrete type like can be, and the empty type can be as well. If a lower bound fails this test the algorithm stops with the answer .

For example, in the problem , we derive that this would be true if were a supertype of . However, is an abstract type, so the relation does not hold.

This concreteness test is done by the function . Note that this test is slightly different from , since it also returns for . Currently this function is heuristic, and does not catch all possible concrete types. The difficulty is that whether a lower bound is concrete might depend on the values of other type variable bounds. For example, is equivalent to the concrete type only if both the upper and lower bounds of equal . We have not yet worked out a complete algorithm for this.

Introduction to the internal machinery

Most operations for dealing with types are found in the files and . A good way to start is to watch subtyping in action. Build Julia with and fire up Julia within a debugger. gdb debugging tips has some tips which may be useful.

Because the subtyping code is used heavily in the REPL itself–and hence breakpoints in this code get triggered often–it will be easiest if you make the following definition:

```

```

and then set a breakpoint in . Once this breakpoint gets triggered, you can set breakpoints in other functions.

As a warm-up, try the following:

```

```

We can make it more interesting by trying a more complex case:

```

```

Subtyping and method sorting

The functions are used for imposing a partial order on functions in method tables (from most-to-least specific). Specificity is strict; if is more specific than , then does not equal and is not more specific than .

If is a strict subtype of , then it is automatically considered more specific. From there, employs some less formal rules. For example, is sensitive to the number of arguments, but may not be. In particular, is more specific than , even though

it is not a subtype. (Of and , neither is more specific than the other.) Likewise, is not a subtype of , but it is considered more specific. However, does get a bonus for length: in particular, is more specific than .

If you're debugging how methods get sorted, it can be convenient to define the function:

which allows you to test whether tuple type is more specific than tuple type .

71.4 Memory layout of Julia Objects

Object layout (jl_value_t)

The struct is the name for a block of memory owned by the Julia Garbage Collector, representing the data associated with a Julia object in memory. Absent any type information, it is simply an opaque pointer:

Each struct is contained in a struct that contains metadata information about the Julia object, such as its type and garbage collector (gc) reachability:

The type of any Julia object is an instance of a leaf object. The function can be used to query for it:

The layout of the object depends on its type. Reflection methods can be used to inspect that layout. A field can be accessed by calling one of the get-field methods:

If the field types are known, a priori, to be all pointers, the values can also be extracted directly as an array access:

As an example, a "boxed" is stored as follows:

This object is created by . Note that the pointer references the data portion, not the metadata at the top of the struct.

A value may be stored "unboxed" in many circumstances (just the data, without the metadata, and possibly not even stored but just kept in registers), so it is unsafe to assume that the address of a box is a unique identifier. The "egal" test (corresponding to the function in Julia), should instead be used to compare two unknown objects for equivalence:

This optimization should be relatively transparent to the API, since the object will be "boxed" on-demand, whenever a pointer is needed.

Note that modification of a pointer in memory is permitted only if the object is mutable. Otherwise, modification of the value may corrupt the program and the result will be undefined. The mutability property of a value can be queried for with:

If the object being stored is a , the Julia garbage collector must be notified also:

However, the Embedding Julia section of the manual is also required reading at this point, for covering other details of boxing and unboxing various types, and understanding the gc interactions.

Mirror structs for some of the built-in types are defined in . The corresponding global objects are created by in .

Garbage collector mark bits

The garbage collector uses several bits from the metadata portion of the to track each object in the system. Further details about this algorithm can be found in the comments of the garbage collector implementation in .

Object allocation

Most new objects are allocated by :

Although, objects can be also constructed directly from memory:

And some objects have special constructors that must be used instead of the above functions:

Types:

While these are the most commonly used options, there are more low-level constructors too, which you can find declared in . These are used in to create the initial types needed to bootstrap the creation of the Julia system image.

Tuples:

The representation of tuples is highly unique in the Julia object representation ecosystem. In some cases, a object may be an array of pointers to the objects contained by the tuple equivalent to:

However, in other cases, the tuple may be converted to an anonymous type and stored unboxed, or it may not stored at all (if it is not being used in a generic context as a).

Symbols:

Functions and MethodInstance:

Arrays:

Note that many of these have alternative allocation functions for various special-purposes. The list here reflects the more common usages, but a more complete list can be found by reading the header file.

Internal to Julia, storage is typically allocated by (or for the special types):

And at the lowest level, memory is getting allocated by a call to the garbage collector (in), then tagged with its type:

Note that all objects are allocated in multiples of 4 bytes and aligned to the platform pointer size. Memory is allocated from a pool for smaller objects, or directly with for large objects.

Singleton Types

Singleton types have only one instance and no data fields. Singleton instances have a size of 0 bytes, and consist only of their metadata. e.g. .

See Singleton Types and Nothingness and missing values

71.5 Eval of Julia code

One of the hardest parts about learning how the Julia Language runs code is learning how all of the pieces work together to execute a block of code.

Each chunk of code typically makes a trip through many steps with potentially unfamiliar names, such as (in no particular order): flisp, AST, C++, LLVM, , , , sysimg (or system image), bootstrapping, compile, parse, execute, JIT, interpret, box, unbox, intrinsic function, and primitive function, before turning into the desired result (hopefully).

Definitions

- REPL

  REPL stands for Read-Eval-Print Loop. It's just what we call the command line environment for short.

- AST
  Abstract Syntax Tree The AST is the digital representation of the code structure. In this form the code has been tokenized for meaning so that it is more suitable for manipulation and execution.

Julia Execution

The 10,000 foot view of the whole process is as follows:

1. The user starts .
2. The C function from gets called. This function processes the command line arguments, filling in the struct and setting the variable . It then initializes Julia (by calling in , which may load a previously compiled sysimg). Finally, it passes off control to Julia by calling .
3. When takes over control, the subsequent sequence of commands depends on the command line arguments given. For example, if a filename was supplied, it will proceed to execute that file. Otherwise, it will start an interactive REPL.
4. Skipping the details about how the REPL interacts with the user, let's just say the program ends up with a block of code that it wants to run.
5. If the block of code to run is in a file, gets invoked to load the file and parse it. Each fragment of code is then passed to to execute.
6. Each fragment of code (or AST), is handed off to to turn into results.
7. takes each code fragment and tries to run it in .
8. decides whether the code is a "toplevel" action (such as or), which would be invalid inside a function. If so, it passes off the code to the toplevel interpreter.
9. then expands the code to eliminate any macros and to "lower" the AST to make it simpler to execute.
10. then uses some simple heuristics to decide whether to JIT compiler the AST or to interpret it directly.
11. The bulk of the work to interpret code is handled by in .
12. If instead, the code is compiled, the bulk of the work is handled by . Whenever a Julia function is called for the first time with a given set of argument types, type inference will be run on that function. This information is used by the codegen step to generate faster code.
13. Eventually, the user quits the REPL, or the end of the program is reached, and the method returns.
14. Just before exiting, calls . This calls (which calls any functions registered to inside Julia). Then it calls . Finally, it gracefully cleans up all handles and waits for them to flush and close.

Parsing

The Julia parser is a small lisp program written in femtolisp, the source-code for which is distributed inside Julia in src/flisp.

The interface functions for this are primarily defined in . The code in handles this handoff on the Julia side.

The other relevant files at this stage are , which handles tokenizing Julia code and turning it into an AST, and , which handles transforming complex AST representations into simpler, "lowered" AST representations which are more suitable for analysis and execution.

Macro Expansion

When encounters a macro, it expands that AST node before attempting to evaluate the expression. Macro expansion involves a handoff from (in Julia), to the parser function (written in) to the Julia macro itself (written in - what else - Julia) via , and back.

Typically, macro expansion is invoked as a first step during a call to /, although it can also be invoked directly by a call to /.

Type Inference

Type inference is implemented in Julia by in. Type inference is the process of examining a Julia function and determining bounds for the types of each of its variables, as well as bounds on the type of the return value from the function. This enables many future optimizations, such as unboxing of known immutable values, and compile-time hoisting of various run-time operations such as computing field offsets and function pointers. Type inference may also include other steps such as constant propagation and inlining.

More Definitions

- JIT
  Just-In-Time Compilation The process of generating native-machine code into memory right when it is needed.
- LLVM
  Low-Level Virtual Machine (a compiler) The Julia JIT compiler is a program/library called libLLVM. Codegen in Julia refers both to the process of taking a Julia AST and turning it into LLVM instructions, and the process of LLVM optimizing that and turning it into native assembly instructions.
- C++
  The programming language that LLVM is implemented in, which means that codegen is also implemented in this language. The rest of Julia's library is implemented in C, in part because its smaller feature set makes it more usable as a cross-language interface layer.
- box
  This term is used to describe the process of taking a value and allocating a wrapper around the data that is tracked by the garbage collector (gc) and is tagged with the object's type.
- unbox
  The reverse of boxing a value. This operation enables more efficient manipulation of data when the type of that data is fully known at compile-time (through type inference).
- generic function
  A Julia function composed of multiple ”methods” that are selected for dynamic dispatch based on the argument type-signature
- anonymous function or ”method”
  A Julia function without a name and without type-dispatch capabilities
- primitive function
  A function implemented in C but exposed in Julia as a named function ”method” (albeit without generic function dispatch capabilities, similar to a anonymous function)
- intrinsic function
  A low-level operation exposed as a function in Julia. These pseudo-functions implement operations on raw bits such as add and sign extend that cannot be expressed directly in any other way. Since they operate on bits directly, they must be compiled into a function and surrounded by a call to to reassign type information to the value.

JIT Code Generation

Codegen is the process of turning a Julia AST into native machine code.

The JIT environment is initialized by an early call to in .

On demand, a Julia method is converted into a native function by the function . (note, when using the MCJIT (in LLVM v3.4+), each function must be JIT into a new module.) This function recursively calls until the entire function has been emitted.

Much of the remaining bulk of this file is devoted to various manual optimizations of specific code patterns. For example, knows how to inline many of the primitive functions (defined in) for various combinations of argument types.

Other parts of codegen are handled by various helper files:

- Handles backtraces for JIT functions
- Handles the ccall and llvmcall FFI, along with various files
- Handles the emission of various low-level intrinsic functions

Bootstrapping

The process of creating a new system image is called "bootstrapping".

The etymology of this word comes from the phrase "pulling oneself up by the bootstraps", and refers to the idea of starting from a very limited set of available functions and definitions and ending with the creation of a full-featured environment.

System Image

The system image is a precompiled archive of a set of Julia files. The file distributed with Julia is one such system image, generated by executing the file , and serializing the resulting environment (including Types, Functions, Modules, and all other defined values) into a file. Therefore, it contains a frozen version of the , , and modules (and whatever else was in the environment at the end of bootstrapping). This serializer/deserializer is implemented by / in .

If there is no sysimg file (), this also implies that was given on the command line, so the final result should be a new sysimg file. During Julia initialization, minimal and modules are created. Then a file named is evaluated from the current directory. Julia then evaluates any file given as a command line argument until it reaches the end. Finally, it saves the resulting environment to a "sysimg" file for use as a starting point for a future Julia run.

71.6 Calling Conventions

Julia uses three calling conventions for four distinct purposes:

Name	Prefix	Purpose
Native		Speed via specialized signatures
JL Call		Wrapper for generic calls
JL Call		Builtins
C ABI		Wrapper callable from C

Julia Native Calling Convention

The native calling convention is designed for fast non-generic calls. It usually uses a specialized signature.

- LLVM ghosts (zero-length types) are omitted.
- LLVM scalars and vectors are passed by value.
- LLVM aggregates (arrays and structs) are passed by reference.

A small return values is returned as LLVM return values. A large return values is returned via the ”structure return” () convention, where the caller provides a pointer to a return slot.

An argument or return values thta is a homogeneous tuple is sometimes represented as an LLVM vector instead of an LLVM array.

JL Call Convention

The JL Call convention is for builtins and generic dispatch. Hand-written functions using this convention are declared via the macro . The convention uses exactly 3 parameters:

- - Julia representation of function that is being applied
- - pointer to array of pointers to boxes
- - length of the array

The return value is a pointer to a box.

C ABI

C ABI wrappers enable calling Julia from C. The wrapper calls a function using the native calling convention.

Tuples are always represented as C arrays.

71.7 High-level Overview of the Native-Code Generation Process

Representation of Pointers

When emitting code to an object file, pointers will be emitted as relocations. The deserialization code will ensure any object that pointed to one of these constants gets recreated and contains the right runtime pointer.

Otherwise, they will be emitted as literal constants.

To emit one of these objects, call . It'll handle tracking the Julia value and the LLVM global, ensuring they are valid both for the current runtime and after deserialization.

When emitted into the object file, these globals are stored as references in a large table. This allows the deserializer to reference them by index, and implement a custom manual mechanism similar to a Global Offset Table (GOT) to restore them.

Function pointers are handled similarly. They are stored as values in a large table. Like globals, this allows the deserializer to reference them by index.

Note that functions are handled separately, with names, via the usual symbol resolution mechanism in the linker.

Note too that functions are also handled separately, via a manual GOT and Procedure Linkage Table (PLT).

Representation of Intermediate Values

Values are passed around in a struct. This represents an R-value, and includes enough information to determine how to assign or pass it somewhere.

They are created via one of the helper constructors, usually: (for immediate values) and (for pointers to values).

The function can transform between any two types. It returns an R-value with set to . It'll cast the object to the requested representation, making heap boxes, allocating stack copies, and computing tagged unions as needed to change the representation.

By contrast will change to , only if it can be done at zero-cost (i.e. without emitting any code).

Union representation

Inferred union types may be stack allocated via a tagged type representation.

The primitive routines that need to be able to handle tagged unions are:

- mark-type
- load-local
- store-local
- isa
- is
- emit_typeof
- emit_sizeof
- boxed
- unbox
- specialized cc-ret

Everything else should be possible to handle in inference by using these primitives to implement union-splitting.

The representation of the tagged-union is as a pair of . The selector is fixed-size as , and will union-tag the first 126 isbits. It records the one-based depth-first count into the type-union of the isbits objects inside. An index of zero indicates that the is actually a tagged heap-allocated , and needs to be treated as normal for a boxed object rather than as a tagged union.

The high bit of the selector () can be tested to determine if the is actually a heap-allocated () box, thus avoiding the cost of re-allocating a box, while maintaining the ability to efficiently handle union-splitting based on the low bits.

It is guaranteed that is an exact test for the type, if the value can be represented by a tag – it will never be marked . It is not necessary to also test the type-tag when testing .

The memory region may be allocated at *any* size. The only constraint is that it is big enough to contain the data currently specified by . It might not be big enough to contain the union of all types that could be stored there according to the associated Union type field. Use appropriate care when copying.

Specialized Calling Convention Signature Representation

A object describes the calling convention details of any callable.

If any of the arguments or return type of a method can be represented unboxed, and the method is not varargs, it'll be given an optimized calling convention signature based on its and fields.

The general principles are that:

- Primitive types get passed in int/float registers.
- Tuples of VecElement types get passed in vector registers.
- Structs get passed on the stack.
- Return values are handle similarly to arguments, with a size-cutoff at which they will instead be returned via a hidden sret argument.

The total logic for this is implemented by and .

Additionally, if the return type is a union, it may be returned as a pair of values (a pointer and a tag). If the union values can be stack-allocated, then sufficient space to store them will also be passed as a hidden first argument. It is up to the callee whether the returned pointer will point to this space, a boxed object, or even other constant memory.

71.8 Julia Functions

This document will explain how functions, method definitions, and method tables work.

Method Tables

Every function in Julia is a generic function. A generic function is conceptually a single function, but consists of many definitions, or methods. The methods of a generic function are stored in a method table. Method tables (type) are associated with s. A describes a family of parameterized types. For example and share the same type name object.

All objects in Julia are potentially callable, because every object has a type, which in turn has a .

Function calls

Given the call , the following steps are performed: first, the method table to use is accessed as . Second, an argument tuple type is formed, . Note that the type of the function itself is the first element. This is because the type might have parameters, and so needs to take part in dispatch. This tuple type is looked up in the method table.

This dispatch process is performed by , which takes two arguments: a pointer to an array of the values f, x, and y, and the number of values (in this case 3).

Throughout the system, there are two kinds of APIs that handle functions and argument lists: those that accept the function and arguments separately, and those that accept a single argument structure. In the first kind of API, the "arguments" part does *not* contain information about the function, since that is passed separately. In the second kind of API, the function is the first element of the argument structure.

For example, the following function for performing a call accepts just an pointer, so the first element of the args array will be the function to call:

This entry point for the same functionality accepts the function separately, so the array does not contain the function:

Adding methods

Given the above dispatch process, conceptually all that is needed to add a new method is (1) a tuple type, and (2) code for the body of the method. implements this operation. is called to extract the relevant method table from what would be the type of the first argument. This is much more complicated than the corresponding procedure during dispatch, since the argument tuple type might be abstract. For example, we can define:

which works since all possible matching methods would belong to the same method table.

Creating generic functions

Since every object is callable, nothing special is needed to create a generic function. Therefore simply creates a new singleton (0 size) subtype of and returns its instance. A function can have a mnemonic "display name" which is used in debug info and when printing objects. For example the name of is . By convention, the name of the created *type* is the same as the function name, with a prepended. So is .

Closures

A closure is simply a callable object with field names corresponding to captured variables. For example, the following code:

is lowered to (roughly):

Constructors

A constructor call is just a call to a type. The type of most types is , so the method table for contains most constructor definitions. One wrinkle is the fallback definition that makes all types callable via :

In this definition the function type is abstract, which is not normally supported. To make this work, all subtypes of (, , , and) currently share a method table via special arrangement.

Builtins

The ”builtin” functions, defined in the module, are:

These are all singleton objects whose types are subtypes of , which is a subtype of . Their purpose is to expose entry points in the run time that use the ”jlcall” calling convention:

The method tables of builtins are empty. Instead, they have a single catch-all method cache entry () whose jlcall fptr points to the correct function. This is kind of a hack but works reasonably well.

Keyword arguments

Keyword arguments work by associating a special, hidden function object with each method table that has definitions with keyword arguments. This function is called the ”keyword argument sorter” or ”keyword sorter”, or ”kwsorter”, and is stored in the field of objects. Every definition in the kwsorter function has the same arguments as some definition in the normal method table, except with a single argument prepended. This array contains alternating symbols and values that represent the passed keyword arguments. The kwsorter’s job is to move keyword arguments into their canonical positions based on name, plus evaluate and substite any needed default value expressions. The result is a normal positional argument list, which is then passed to yet another function.

The easiest way to understand the process is to look at how a keyword argument method definition is lowered. The code:

actually produces *three* method definitions. The first is a function that accepts all arguments (including keywords) as positional arguments, and includes the code for the method body. It has an auto-generated name:

The second method is an ordinary definition for the original function, which handles the case where no keyword arguments are passed:

This simply dispatches to the first method, passing along default values. Finally there is the kwsorter definition:

The front end generates code to loop over the array and pick out arguments in the right order, evaluating default expressions when an argument is not found.

The function fetches (and creates, if necessary) the field .

This design has the feature that call sites that don't use keyword arguments require no special handling; everything works as if they were not part of the language at all. Call sites that do use keyword arguments are dispatched directly to the called function's kwsorter. For example the call:

is lowered to:

The unpacking procedure represented here as actually further unpacks each *element* of , expecting each one to contain two values (a symbol and a value). (also in) fetches the kwsorter for the called function. Notice that the original function is passed through, to handle closures.

Compiler efficiency issues

Generating a new type for every function has potentially serious consequences for compiler resource use when combined with Julia's "specialize on all arguments by default" design. Indeed, the initial implementation of this design suffered from much longer build and test times, higher memory use, and a system image nearly 2x larger than the baseline. In a naive implementation, the problem is bad enough to make the system nearly unusable. Several significant optimizations were needed to make the design practical.

The first issue is excessive specialization of functions for different values of function-valued arguments. Many functions simply "pass through" an argument to somewhere else, e.g. to another function or to a storage location. Such functions do not need to be specialized for every closure that might be passed in. Fortunately this case is easy to distinguish by simply considering whether a function *calls* one of its arguments (i.e. the argument appears in "head position" somewhere). Performance-critical higher-order functions like certainly call their argument function and so will still be specialized as expected. This optimization is implemented by recording which arguments are called during the pass in the front end. When sees an argument in the type hierarchy passed to a slot declared as or , it behaves as if the annotation were applied. This heuristic seems to be extremely effective in practice.

The next issue concerns the structure of method cache hash tables. Empirical studies show that the vast majority of dynamically-dispatched calls involve one or two arguments. In turn, many of these cases can be resolved by considering only the first argument. (Aside: proponents of single dispatch would not be surprised by this at all. However, this argument means "multiple dispatch is easy to optimize in practice", and that we should therefore use it, *not* "we should use single dispatch"!) So the method cache uses the type of the first argument as its primary key. Note, however, that this corresponds to the *second* element of the tuple type for a function call (the first element being the type of the function itself). Typically, type variation in head position is extremely low – indeed, the majority of functions belong to singleton types with no parameters. However, this is not the case for constructors, where a single method table holds constructors for every type. Therefore the method table is special-cased to use the *first* tuple type element instead of the second.

The front end generates type declarations for all closures. Initially, this was implemented by generating normal type declarations. However, this produced an extremely large number of constructors, all of which were trivial (simply passing all arguments through to). Since methods are partially ordered, inserting all of these methods is O(n^2), plus there are just too many of them to keep around. This was optimized by generating expressions directly (bypassing default constructor generation), and using directly to create closure instances. Not the prettiest thing ever, but you do what you gotta do.

The next problem was the macro, which generated a 0-argument closure for each test case. This is not really necessary, since each test case is simply run once in place. Therefore I modified to expand to a try-catch block that records the test result (true, false, or exception raised) and calls the test suite handler on it.

71.9 Base.Cartesian

The (non-exported) Cartesian module provides macros that facilitate writing multidimensional algorithms. It is hoped that Cartesian will not, in the long term, be necessary; however, at present it is one of the few ways to write compact and performant multidimensional code.

Principles of usage

A simple example of usage is:

which generates the following code:

In general, Cartesian allows you to write generic code that contains repetitive elements, like the nested loops in this example. Other applications include repeated expressions (e.g., loop unwinding) or creating function calls with variable numbers of arguments without using the ”splat” construct ().

Basic syntax

The (basic) syntax of is as follows:

- The first argument must be an integer (*not* a variable) specifying the number of loops.
- The second argument is the symbol-prefix used for the iterator variable. Here we used , and variables were generated.
- The third argument specifies the range for each iterator variable. If you use a variable (symbol) here, it’s taken as . More flexibly, you can use the anonymous-function expression syntax described below.
- The last argument is the body of the loop. Here, that’s what appears between the .

There are some additional features of described in the reference section.

 follows a similar pattern, generating from . The general practice is to read from left to right, which is why is (as in , where is to the left and the range is to the right) whereas is (as in , where the array comes first).

If you’re developing code with Cartesian, you may find that debugging is easier when you examine the generated code, using :

Supplying the number of expressions

The first argument to both of these macros is the number of expressions, which must be an integer. When you're writing a function that you intend to work in multiple dimensions, this may not be something you want to hard-code. If you're writing code that you need to work with older Julia versions, currently you should use the macro described in an older version of this documentation.

Starting in Julia 0.4-pre, the recommended approach is to use a . Here's an example:

Naturally, you can also prepare expressions or perform calculations before the block.

Anonymous-function expressions as macro arguments

Perhaps the single most powerful feature in is the ability to supply anonymous-function expressions that get evaluated at parsing time. Let's consider a simple example:

 generates expressions that follow a pattern. This code would generate the following statements:

In each generated statement, an "isolated" (the variable of the anonymous function) gets replaced by values in the range . Generally speaking, Cartesian employs a LaTeX-like syntax. This allows you to do math on the index . Here's an example computing the strides of an array:

would generate expressions

Anonymous-function expressions have many uses in practice.

Macro reference – Macro.

Generate nested loops, using as the prefix for the iteration variables. may be an anonymous-function expression, or a simple symbol in which case the range is for dimension .

Optionally, you can provide ”pre” and ”post” expressions. These get executed first and last, respectively, in the body of each loop. For example:

would generate:

If you want just a post-expression, supply for the pre-expression. Using parentheses and semicolons, you can supply multi-statement expressions.

– Macro.

Generate expressions like . can either be an iteration-symbol prefix, or an anonymous-function expression.

Examples

– Macro.

Generate variables , , ..., to extract values from . can be either a or anonymous-function expression.

would generate

while yields

– Macro.

Generate expressions. should be an anonymous-function expression.

Examples

– Macro.

Generate a function call expression. represents any number of function arguments, the last of which may be an anonymous-function expression and is expanded into arguments.

For example generates

while yields

– Macro.

Generates an -tuple. would generate , and would generate .

– Macro.

Check whether all of the expressions generated by the anonymous-function expression evaluate to .

would generate the expression . This can be convenient for bounds-checking.

– Macro.

Check whether any of the expressions generated by the anonymous-function expression evaluate to .

would generate the expression .

– Macro.

Generates a sequence of statements. For example:

would generate:

71.10 Talking to the compiler (the mechanism)

In some circumstances, one might wish to provide hints or instructions that a given block of code has special properties: you might always want to inline it, or you might want to turn on special compiler optimization passes. Starting with version 0.4, Julia has a convention that these instructions can be placed inside a expression, which is typically (but not necessarily) the first expression in the body of a function.

expressions are created with macros. As an example, consider the implementation of the macro:

Here, is expected to be an expression defining a function. A statement like this:

gets turned into an expression like this:

appends to the end of the expression, creating a new expression if necessary. If is specified, a nested expression containing and these arguments is appended instead, which can be used to specify additional information.

To use the metadata, you have to parse these expressions. If your implementation can be performed within Julia, is very handy: will scan a function *body* expression (one without the function signature) for the first expression containing , extract any arguments, and return a tuple . If the metadata did not have any arguments, or was not found, the array will be empty.

Not yet provided is a convenient infrastructure for parsing expressions from C++.

71.11 SubArrays

Julia's type is a container encoding a "view" of a parent . This page documents some of the design principles and implementation of s.

Indexing: cartesian vs. linear indexing

Broadly speaking, there are two main ways to access data in an array. The first, often called cartesian indexing, uses indexes for an -dimensional . For example, a matrix (2-dimensional) can be indexed in cartesian style as . The second indexing method, referred to as linear indexing, uses a single index even for higher-dimensional objects. For example, if , then the expression returns the value 5. Julia allows you to combine these styles of indexing: for example, a 3d array can be indexed as , in which case is interpreted as a cartesian index for the first dimension, and is a linear index over dimensions 2 and 3.

For s, linear indexing appeals to the underlying storage format: an array is laid out as a contiguous block of memory, and hence the linear index is just the offset (+1) of the corresponding entry relative to the beginning of the array. However, this is not true for many other types: examples include , arrays that require some kind of computation (such as interpolation), and the type under discussion here, . For these types, the underlying information is more naturally described in terms of cartesian indexes.

You can manually convert from a cartesian index to a linear index with , and vice versa using . and functions for types may include similar operations.

While converting from a cartesian index to a linear index is fast (it's just multiplication and addition), converting from a linear index to a cartesian index is very slow: it relies on the operation, which is one of the slowest low-level operations you can perform with a CPU. For this reason, any code that deals with types is best designed in terms of cartesian, rather than linear, indexing.

Index replacement

Consider making 2d slices of a 3d array:

drops "singleton" dimensions (ones that are specified by an), so both and are two-dimensional s. Consequently, the natural way to index these is with . To extract the value from the parent array , the natural approach is to replace with and with .

The key feature of the design of SubArrays is that this index replacement can be performed without any runtime overhead.

SubArray design

Type parameters and fields

The strategy adopted is first and foremost expressed in the definition of the type:

has 5 type parameters. The first two are the standard element type and dimensionality. The next is the type of the parent . The most heavily-used is the fourth parameter, a of the types of the indices for each dimension. The final one, , is only provided as a convenience for dispatch; it's a boolean that represents whether the index types support fast linear indexing. More on that later.

If in our example above is a , our case above would be a . Note in particular the tuple parameter, which stores the types of the indices used to create . Likewise,

Storing these values allows index replacement, and having the types encoded as parameters allows one to dispatch to efficient algorithms.

Index translation

Performing index translation requires that you do different things for different concrete types. For example, for , one needs to apply the indices to the first and third dimensions of the parent array, whereas for one needs to apply them to the second and third. The simplest approach to indexing would be to do the type-analysis at runtime:

Unfortunately, this would be disastrous in terms of performance: each element access would allocate memory, and involves the running of a lot of poorly-typed code.

The better approach is to dispatch to specific methods to handle each type of stored index. That's what does: it dispatches on the type of the first stored index and consumes the appropriate number of input indices, and then it recurses on the remaining indices. In the case of , this expands to

for any pair of indices (except s and arrays thereof, see below).

This is the core of a ; indexing methods depend upon to do this index translation. Sometimes, though, we can avoid the indirection and make it even faster.

Linear indexing

Linear indexing can be implemented efficiently when the entire array has a single stride that separates successive elements, starting from some offset. This means that we can pre-compute these values and represent linear indexing simply as an addition and multiplication, avoiding the indirection of and (more importantly) the slow computation of the cartesian coordinates entirely.

For types, the availability of efficient linear indexing is based purely on the types of the indices, and does not depend on values like the size of the parent array. You can ask whether a given set of indices supports fast linear indexing with the internal function:

This is computed during construction of the and stored in the type parameter as a boolean that encodes fast linear indexing support. While not strictly necessary, it means that we can define dispatch directly on without any intermediaries.

Since this computation doesn't depend on runtime values, it can miss some cases in which the stride happens to be uniform:

A view constructed as happens to have uniform stride, and therefore linear indexing indeed could be performed efficiently. However, success in this case depends on the size of the array: if the first dimension instead were odd,

then does not have uniform stride, so we cannot guarantee efficient linear indexing. Since we have to base this decision based purely on types encoded in the parameters of the , cannot implement efficient linear indexing.

A few details

- Note that the function is agnostic to the types of the input indices; it simply determines how and where the stored indices should be reindexed. It not only supports integer indices, but it supports non-scalar indexing, too. This means that views of views don't need two levels of indirection; they can simply re-compute the indices into the original parent array!
- Hopefully by now it's fairly clear that supporting slices means that the dimensionality, given by the parameter , is not necessarily equal to the dimensionality of the parent array or the length of the tuple. Neither do user-supplied indices necessarily line up with entries in the tuple (e.g., the second user-supplied index might correspond to the third dimension of the parent array, and the third element in the tuple).

  What might be less obvious is that the dimensionality of the stored parent array must be equal to the number of effective indices in the tuple. Some examples:

  Naively, you'd think you could just set and , but supporting this dramatically complicates the reindexing process, especially for views of views. Not only do you need to dispatch on the types of the stored indices, but you need to examine whether a given index is the final one and "merge" any remaining stored indices together. This is not an easy task, and even worse: it's slow since it implicitly depends upon linear indexing.

  Fortunately, this is precisely the computation that performs, and it does so linearly if possible. Consequently, ensures that the parent array is the appropriate dimensionality for the given indices by reshaping it if needed. The inner constructor ensures that this invariant is satisfied.

- and arrays thereof throw a nasty wrench into the scheme. Recall that simply dispatches on the type of the stored indices in order to determine how many passed indices should be used and where they should go. But with , there's no longer a one-to-one correspondence between the number of passed arguments and the number of dimensions that they index into. If we return to the above example of , you can see that the expansion is incorrect for . It should *skip* the entirely and return:

  Instead, though, we get:

Doing this correctly would require *combined* dispatch on both the stored and passed indices across all combinations of dimensionalities in an intractable manner. As such, must never be called with indices. Fortunately, the scalar case is easily handled by first flattening the arguments to plain integers. Arrays of , however, cannot be split apart into orthogonal pieces so easily. Before attempting to use , must ensure that there are no arrays of in the argument list. If there are, it can simply "punt" by avoiding the calculation entirely, constructing a nested with two levels of indirection instead.

71.12 System Image Building

Building the Julia system image

Julia ships with a preparsed system image containing the contents of the module, named . This file is also precompiled into a shared library called on as many platforms as possible, so as to give vastly improved startup times. On systems that do not ship with a precompiled system image file, one can be generated from the source files shipped in Julia's folder.

This operation is useful for multiple reasons. A user may:

- Build a precompiled shared library system image on a platform that did not ship with one, thereby improving startup times.
- Modify , rebuild the system image and use the new next time Julia is started.
- Include a file that includes packages into the system image, thereby creating a system image that has packages embedded into the startup environment.

Julia now ships with a script that automates the tasks of building the system image, wittingly named that lives in . That is, to include it into a current Julia session, type:

This will include a function:

– Function.

Rebuild the system image. Store it in , which defaults to a file named that sits in the same folder as , except on Windows where it defaults to . Use the cpu instruction set given by . Valid CPU targets are the same as for the option to , or the option to . Defaults to , which means to use all CPU instructions available on the current processor. Include the user image file given by , which should contain directives such as to include that package in the new system image. New system image will not replace an older image unless is set to true.

Note that this file can also be run as a script itself, with command line arguments taking the place of arguments passed to the function. For example, to build a system image in , with the CPU instruction set, a user image of and set to , one would execute:

71.13 Working with LLVM

This is not a replacement for the LLVM documentation, but a collection of tips for working on LLVM for Julia.

Overview of Julia to LLVM Interface

Julia statically links in LLVM by default. Build with to link dynamically.

The code for lowering Julia AST to LLVM IR or interpreting it directly is in directory .

File	Description
	Builtin functions
	Lowering
	Lowering utilities, notably for array and tuple accesses
	Top-level of code generation, pass list, lowering builtins
	Tracks debug information for JIT code
	Handles native object file and JIT code diassembly
	Generic functions
	Lowering intrinsics
	Custom LLVM pass for
	I/O and operating system utility functions

Some of the files form a group that compile to a single object.

The difference between an intrinsic and a builtin is that a builtin is a first class function that can be used like any other Julia function. An intrinsic can operate only on unboxed data, and therefore its arguments must be statically typed.

Alias Analysis

Julia currently uses LLVM's Type Based Alias Analysis. To find the comments that document the inclusion relationships, look for in .

The option enables LLVM's Basic Alias Analysis.

Building Julia with a different version of LLVM

The default version of LLVM is specified in . You can override it by creating a file called in the top-level directory and adding a line to it such as:

Besides the LLVM release numerals, you can also use to bulid against the latest development version of LLVM.

Passing options to LLVM

You can pass options to LLVM using *debug* builds of Julia. To create a debug build, run . The resulting executable is . You can pass LLVM options to this executable via the environment variable . Here are example settings using syntax:

- dumps IR after each pass.
- dumps LLVM diagnostics for loop vectorizer *if* you built Julia with . Otherwise you will get warnings about "Unknown command line argument". Counter-intuitively, building Julia with is *not* enough to dump diagnostics from a pass.

Debugging LLVM transformations in isolation

On occasion, it can be useful to debug LLVM's transformations in isolation from the rest of the Julia system, e.g. because reproducing the issue inside would take too long, or because one wants to take advantage of LLVM's tooling (e.g. bugpoint). To get unoptimized IR for the entire system image, pass the option to the system image build process, which

will output the unoptimized IR to an file. This file can then be passed to LLVM tools as usual. can function as an LLVM pass plugin and can be loaded into LLVM tools, to make julia-specific passes available in this environment. In addition, it exposes the meta-pass, which runs the entire Julia pass-pipeline over the IR. As an example, to generate a system image, one could do:

This system image can then be loaded by as usual.

Alternatively, you can use to obtain a trace of all IR passed to the JIT. This is useful for code that cannot be run as part of the sysimg generation process (e.g. because it creates unserializable state). However, the resulting does not include sysimage data, and can thus not be used as such.

It is also possible to dump an LLVM IR module for just one Julia function, using:

These files can be processed the same way as the unoptimized sysimg IR shown above.

Improving LLVM optimizations for Julia

Improving LLVM code generation usually involves either changing Julia lowering to be more friendly to LLVM's passes, or improving a pass.

If you are planning to improve a pass, be sure to read the LLVM developer policy. The best strategy is to create a code example in a form where you can use LLVM's tool to study it and the pass of interest in isolation.

1. Create an example Julia code of interest.
2. Use to dump the IR.
3. Pick out the IR at the point just before the pass of interest runs.
4. Strip the debug metadata and fix up the TBAA metadata by hand.

The last step is labor intensive. Suggestions on a better way would be appreciated.

The jlcall calling convention

Julia has a generic calling convention for unoptimized code, which looks somewhat as follows:

where the first argument is the boxed function object, the second argument is an on-stack array of arguments and the third is the number of arguments. Now, we could perform a straightforward lowering and emit an alloca for the argument array. However, this would betray the SSA nature of the uses at the call site, making optimizations (including GC root placement), significantly harder. Instead, we emit it as follows:

The special annotation marks the fact that this call site is really using the jlcall calling convention. This allows us to retain the SSA-ness of the uses throughout the optimizer. GC root placement will later lower this call to the original C ABI. In the code the calling convention number is represented by the constant. In addition, there is the calling convention which functions similarly, but omits the first argument.

GC root placement

GC root placement is done by an LLVM pass late in the pass pipeline. Doing GC root placement this late enables LLVM to make more aggressive optimizations around code that requires GC roots, as well as allowing us to reduce the number of required GC roots and GC root store operations (since LLVM doesn't understand our GC, it wouldn't otherwise know what it is and is not allowed to do with values stored to the GC frame, so it'll conservatively do very little). As an example, consider an error path

During constant folding, LLVM may discover that the condition is always false, and can remove the basic block. However, if GC root lowering is done early, the GC root slots used in the deleted block, as well as any values kept alive in those slots only because they were used in the error path, would be kept alive by LLVM. By doing GC root lowering late, we give LLVM the license to do any of its usual optimizations (constant folding, dead code elimination, etc.), without having to worry (too much) about which values may or may not be GC tracked.

However, in order to be able to do late GC root placement, we need to be able to identify a) which pointers are gc tracked and b) all uses of such pointers. The goal of the GC placement pass is thus simple:

Minimize the number of needed GC roots/stores to them subject to the constraint that at every safepoint, any live GC-tracked pointer (i.e. for which there is a path after this point that contains a use of this pointer) is in some GC slot.

Representation

The primary difficulty is thus choosing an IR representation that allows us to identify GC-tracked pointers and their uses, even after the program has been run through the optimizer. Our design makes use of three LLVM features to achieve this:

- Custom address spaces
- Operand Bundles
- Non-integral pointers

Custom address spaces allow us to tag every point with an integer that needs to be preserved through optimizations. The compiler may not insert casts between address spaces that did not exist in the original program and it must never change the address space of a pointer on a load/store/etc operation. This allows us to annotate which pointers are GC-tracked in an optimizer-resistant way. Note that metadata would not be able to achieve the same purpose. Metadata is supposed to always be discardable without altering the semantics of the program. However, failing to identify a GC-tracked pointer alters the resulting program behavior dramatically - it'll probably crash or return wrong results. We currently use three different address spaces (their numbers are defined in):

- GC Tracked Pointers (currently 10): These are pointers to boxed values that may be put into a GC frame. It is loosely equivalent to a pointer on the C side. N.B. It is illegal to ever have a pointer in this address space that may not be stored to a GC slot.

- Derived Pointers (currently 11): These are pointers that are derived from some GC tracked pointer. Uses of these pointers generate uses of the original pointer. However, they need not themselves be known to the GC. The GC root placement pass MUST always find the GC tracked pointer from which this pointer is derived and use that as the pointer to root.
- Callee Rooted Pointers (currently 12): This is a utility address space to express the notion of a callee rooted value. All values of this address space MUST be storable to a GC root (though it is possible to relax this condition in the future), but unlike the other pointers need not be rooted if passed to a call (they do still need to be rooted if they are live across another safepoint between the definition and the call).

Invariants

The GC root placement pass makes use of several invariants, which need to be observed by the frontend and are preserved by the optimizer.

First, only the following address space casts are allowed:

- 0->{Tracked,Derived,CalleeRooted}: It is allowable to decay an untracked pointer to any of the others. However, do note that the optimizer has broad license to not root such a value. It is never safe to have a value in address space 0 in any part of the program if it is (or is derived from) a value that requires a GC root.
- Tracked->Derived: This is the standard decay route for interior values. The placement pass will look for these to identify the base pointer for any use.
- Tracked->CalleeRooted: Addrspace CalleeRooted serves merely as a hint that a GC root is not required. However, do note that the Derived->CalleeRooted decay is prohibited, since pointers should generally be storable to a GC slot, even in this address space.

Now let us consider what constitutes a use:

- Loads whose loaded values is in one of the address spaces
- Stores of a value in one of the address spaces to a location
- Stores to a pointer in one of the address spaces
- Calls for which a value in one of the address spaces is an operand
- Calls in jlcall ABI, for which the argument array contains a value
- Return instructions.

We explicitly allow load/stores and simple calls in address spaces Tracked/Derived. Elements of jlcall argument arrays must always be in address space Tracked (it is required by the ABI that they are valid pointers). The same is true for return instructions (though note that struct return arguments are allowed to have any of the address spaces). The only allowable use of an address space CalleeRooted pointer is to pass it to a call (which must have an appropriately typed operand).

Further, we disallow in addrspace Tracked. This is because unless the operation is a noop, the resulting pointer will not be validly storable to a GC slot and may thus not be in this address space. If such a pointer is required, it should be decayed to addrspace Derived first.

Lastly, we disallow / instructions in these address spaces. Having these instructions would mean that some values are really GC tracked. This is problematic, because it breaks that stated requirement that we're able to identify GC-relevant pointers. This invariant is accomplished using the LLVM "non-integral pointers" feature, which is new in LLVM 5.0. It prohibits the optimizer from making optimizations that would introduce these operations. Note we can still insert static constants at JIT time by using in address space 0 and then decaying to the appropriate address space afterwards.

Supporting ccall

One important aspect missing from the discussion so far is the handling of . has the peculiar feature that the location and scope of a use do not coincide. As an example consider:

In lowering, the compiler will insert a conversion from the array to the pointer which drops the reference to the array value. However, we of course need to make sure that the array does stay alive while we're doing the . To understand how this is done, first recall the lowering of the above code:

The last , is an extra argument list inserted during lowering that informs the code generator which Julia level values need to be kept alive for the duration of this . We then take this information and represent it in an "operand bundle" at the IR level. An operand bundle is essentially a fake use that is attached to the call site. At the IR level, this looks like so:

The GC root placement pass will treat the operand bundle as if it were a regular operand. However, as a final step, after the GC roots are inserted, it will drop the operand bundle to avoid confusing instruction selection.

Supporting pointer_from_objref

is special because it requires the user to take explicit control of GC rooting. By our above invariants, this function is illegal, because it performs an address space cast from 10 to 0. However, it can be useful, in certain situations, so we provide a special intrinsic:

which is lowered to the corresponding address space cast after GC root lowering. Do note however that by using this intrinsic, the caller assumes all responsibility for making sure that the value in question is rooted. Further this intrinsic is not considered a use, so the GC root placement pass will not provide a GC root for the function. As a result, the external rooting must be arranged while the value is still tracked by the system. I.e. it is not valid to attempt to use the result of this operation to establish a global root - the optimizer may have already dropped the value.

71.14 printf() and stdio in the Julia runtime

Libuv wrappers for stdio

defines libuv wrappers for the streams:

... and corresponding output functions:

These functions are used by the files in the and directories wherever stdio is needed to ensure that output buffering is handled in a unified way.

In special cases, like signal handlers, where the full libuv infrastructure is too heavy, can be used to directly to :

Interface between JL_STD* and Julia code

, and are bound to the libuv streams defined in the runtime.

Julia's function (in) calls (in) to create Julia objects for , and .

uses to retrieve pointers to and calls to inspect the type of each stream. It then creates a Julia , or object to represent each stream, e.g.:

The and methods for these streams use to call libuv wrappers in , e.g.:

printf() during initialization

The libuv streams relied upon by etc., are not available until midway through initialization of the runtime (see ,). Error messages or warnings that need to be printed before this are routed to the standard C library function by the following mechanism:

In , the stream pointers are statically initialized to integer constants: . In the function checks its argument and calls if stream is set to or .

This allows for uniform use of throughout the runtime regardless of whether or not any particular piece of code is reachable before initialization is complete.

Legacy library

The library is inherited from femtolisp. It provides cross-platform buffered file IO and in-memory temporary buffers.

is still used by:

-
- – for serialization file IO and for memory buffers.
- – for serialization file IO and for memory buffers.
- – for file IO (see for libuv equivalent).

Use of in these modules is mostly self-contained and separated from the libuv I/O system. However, there is one place where femtolisp calls through to with a legacy stream.

There is a hack in that makes the field line up with the and ensures that the values used for to not overlap with valid values. This allows pointers to point to streams.

This is needed because caller is passed an stream by femtolisp's function. Julia's function has special handling for this:

71.15 Bounds checking

Like many modern programming languages, Julia uses bounds checking to ensure program safety when accessing arrays. In tight inner loops or other performance critical situations, you may wish to skip these bounds checks to improve runtime performance. For instance, in order to emit vectorized (SIMD) instructions, your loop body cannot contain branches, and thus cannot contain bounds checks. Consequently, Julia includes an macro to tell the compiler to skip such bounds checks within the given block. For the built-in type, the magic happens inside the and intrinsics. User-defined array types instead use the macro to achieve context-sensitive code selection.

Eliding bounds checks

The macro marks blocks of code that perform bounds checking. When such blocks appear inside of an block, the compiler removes these blocks. When the is nested inside of a calling function containing an , the compiler will remove the block *only if it is inlined* into the calling function. For example, you might write the method as:

With a custom array-like type having:

Then when is inlined into , the call to will be elided. If your function contains multiple layers of inlining, only blocks at most one level of inlining deeper are eliminated. The rule prevents unintended changes in program behavior from code further up the stack.

Propagating inbounds

There may be certain scenarios where for code-organization reasons you want more than one layer between the and declarations. For instance, the default methods have the chain calls calls .

To override the ”one layer of inlining” rule, a function may be marked with to propagate an inbounds context (or out of bounds context) through one additional layer of inlining.

The bounds checking call hierarchy

The overall hierarchy is:

- which calls
  - which calls
    - which recursively calls

- for each dimension

Here is the array, and contains the ”requested” indices. returns a tuple of ”permitted” indices of .

throws an error if the indices are invalid, whereas returns in that circumstance. discards any information about the array other than its tuple, and performs a pure indices-vs-indices comparison: this allows relatively few compiled methods to serve a huge variety of array types. Indices are specified as tuples, and are usually compared in a 1-1 fashion with individual dimensions handled by calling another important function, : typically,

```

```

so checks a single dimension. All of these functions, including the unexported have docstrings accessible with .

If you have to customize bounds checking for a specific array type, you should specialize . However, in most cases you should be able to rely on as long as you supply useful for your array type.

If you have novel index types, first consider specializing , which handles a single index for a particular dimension of an array. If you have a custom multidimensional index type (similar to), then you may have to consider specializing .

Note this hierarchy has been designed to reduce the likelihood of method ambiguities. We try to make the place to specialize on array type, and try to avoid specializations on index types; conversely, is intended to be specialized only on index type (especially, the last argument).

71.16 Proper maintenance and care of multi-threading locks

The following strategies are used to ensure that the code is dead-lock free (generally by addressing the 4th Coffman condition: circular wait).

1. structure code such that only one lock will need to be acquired at a time
2. always acquire shared locks in the same order, as given by the table below
3. avoid constructs that expect to need unrestricted recursion

Locks

Below are all of the locks that exist in the system and the mechanisms for using them that avoid the potential for dead-locks (no Ostrich algorithm allowed here):

The following are definitely leaf locks (level 1), and must not try to acquire any other lock:

- safepoint

  Note that this lock is acquired implicitly by and . use the variants to avoid that for level 1 locks.
  While holding this lock, the code must not do any allocation or hit any safepoints. Note that there are safepoints when doing allocation, enabling / disabling GC, entering / restoring exception frames, and taking / releasing locks.
- shared_map
- finalizers
- pagealloc
- gc_perm_lock

- flisp

  flisp itself is already threadsafe, this lock only protects the pool

The following is a leaf lock (level 2), and only acquires level 1 locks (safepoint) internally:

- typecache

The following is a level 3 lock, which can only acquire level 1 or level 2 locks internally:

- Method->writelock

The following is a level 4 lock, which can only recurse to acquire level 1, 2, or 3 locks:

- MethodTable->writelock

No Julia code may be called while holding a lock above this point.

The following is a level 6 lock, which can only recurse to acquire locks at lower levels:

- codegen

The following is an almost root lock (level end-1), meaning only the root look may be held when trying to acquire it:

- typeinf

  this one is perhaps one of the most tricky ones, since type-inference can be invoked from many points
  currently the lock is merged with the codegen lock, since they call each other recursively

The following is the root lock, meaning no other lock shall be held when trying to acquire it:

- toplevel

  this should be held while attempting a top-level action (such as making a new type or defining a new method): trying to obtain this lock inside a staged function will cause a deadlock condition!
  additionally, it's unclear if *any* code can safely run in parallel with an arbitrary toplevel expression, so it may require all threads to get to a safepoint first

Broken Locks

The following locks are broken:

- toplevel

  doesn't exist right now
  fix: create it

Shared Global Data Structures

These data structures each need locks due to being shared mutable global state. It is the inverse list for the above lock priority list. This list does not include level 1 leaf resources due to their simplicity.

MethodTable modifications (def, cache, kwsorter type) : MethodTable->writelock

Type declarations : toplevel lock

Type application : typecache lock

Module serializer : toplevel lock

JIT & type-inference : codegen lock

MethodInstance updates : codegen lock

- These fields are generally lazy initialized, using the test-and-test-and-set pattern.
- These are set at construction and immutable:
  - specTypes
  - sparam_vals
  - def
- These are set by (while holding codegen lock):
  - rettype
  - inferred
  - these can also be reset, see for that logic as it needs to keep in sync
- flag:
  - optimization to quickly avoid recurring into while it is already running
  - actual state (of setting , then) is protected by codegen lock
- Function pointers (and ,):
  - these transition once, from to a value, while the codegen lock is held
- Code-generator cache (the contents of):
  - these can transition multiple times, but only while the codegen lock is held
  - it is valid to use old version of this, or block for new versions of this, so races are benign, as long as the code is careful not to reference other data in the method instance (such as) and assume it is coordinated, unless also holding the codegen lock
- flag:
  - unknown

LLVMContext : codegen lock

Method : Method->writelock

- roots array (serializer and codegen)
- invoke / specializations / tfunc modifications

71.17 Arrays with custom indices

Julia 0.5 adds experimental support for arrays with arbitrary indices. Conventionally, Julia's arrays are indexed starting at 1, whereas some other languages start numbering at 0, and yet others (e.g., Fortran) allow you to specify arbitrary starting indices. While there is much merit in picking a standard (i.e., 1 for Julia), there are some algorithms which simplify considerably if you can index outside the range (and not just , either). Such array types are expected to be supplied through packages.

The purpose of this page is to address the question, "what do I have to do to support such arrays in my own code?" First, let's address the simplest case: if you know that your code will never need to handle arrays with unconventional indexing, hopefully the answer is "nothing." Old code, on conventional arrays, should function essentially without alteration as long as it was using the exported interfaces of Julia.

Generalizing existing code

As an overview, the steps are:

- replace many uses of with
- replace with , and with
- replace explicit allocations like with

These are described in more detail below.

Background

Because unconventional indexing breaks deeply-held assumptions throughout the Julia ecosystem, early adopters running code that has not been updated are likely to experience errors. The most frustrating bugs would be incorrect results or segfaults (total crashes of Julia). For example, consider the following function:

This code implicitly assumes that vectors are indexed from 1. Previously that was a safe assumption, so this code was fine, but (depending on what types the user passes to this function) it may no longer be safe. If this code continued to work when passed a vector with non-1 indices, it would either produce an incorrect answer or it would segfault. (If you do get segfaults, to help locate the cause try running julia with the option .)

To ensure that such errors are caught, in Julia 0.5 both and **should** throw an error when passed an array with non-1 indexing. This is designed to force users of such arrays to check the code, and inspect it for whether it needs to be generalized.

Using for bounds checks and loop iteration

(reminiscent of) returns a tuple of objects, specifying the range of valid indices along each dimension of . When has unconventional indexing, the ranges may not start at 1. If you just want the range for a particular dimension , there is .

Base implements a custom range type, , where means the same thing as but in a form that guarantees (via the type system) that the lower index is 1. For any new type, this is the default returned by , and it indicates that this array type uses "conventional" 1-based indexing. Note that if you don't want to be bothered supporting arrays with non-1 indexing, you can add the following line:

at the top of any function.

For bounds checking, note that there are dedicated functions and which can sometimes simplify such tests.

Linear indexing ()

Some algorithms are most conveniently (or efficiently) written in terms of a single linear index, even if is multi-dimensional. In "true" linear indexing, the indices always range from . However, this raises an ambiguity for one-dimensional arrays (a.k.a.,): does mean linear indexing, or Cartesian indexing with the array's native indices?

For this reason, if you want to use linear indexing in an algorithm, your best option is to get the index range by calling . This will return if is an , and the equivalent of otherwise.

In a sense, one can say that 1-dimensional arrays always use Cartesian indexing. To help enforce this, it's worth noting that and will throw an error if indicates a 1-dimensional array with unconventional indexing (i.e., is a rather than a tuple of). For arrays with conventional indexing, these functions continue to work the same as always.

Using and , here is one way you could rewrite :

Allocating storage using generalizations of

Storage is often allocated with or . When the result needs to match the indices of some other array, this may not always suffice. The generic replacement for such patterns is to use . indicates the kind of underlying "conventional" behavior you'd like, e.g., or or even (which would allocate an all-zeros array). is a tuple of or values, specifying the indices that you want the result to use.

Let's walk through a couple of explicit examples. First, if has conventional indices, then would end up calling , and thus return an array. If is an type with unconventional indexing, then should return something that "behaves like" an but with a shape (including indices) that matches . (The most obvious implementation is to allocate an and then "wrap" it in a type that shifts the indices.)

Note also that would allocate an (i.e., 1-dimensional array) that matches the indices of the columns of .

Deprecations

In generalizing Julia's code base, at least one deprecation was unavoidable: earlier versions of Julia defined , meaning that the first index along a dimension indexed by is 1. This definition can no longer be justified, so it was deprecated. There is no provided replacement, because the proper replacement depends on what you are doing and might need to know more about the array. However, it appears that many uses of are really about computing an index offset; when that is the case, a candidate replacement is:

In other words, while does not itself make sense, in general you can say that the offset associated with a colon-index is zero.

Writing custom array types with non-1 indexing

Most of the methods you'll need to define are standard for any type, see Abstract Arrays. This page focuses on the steps needed to define unconventional indexing.

Do not implement or

Perhaps the majority of pre-existing code that uses will not work properly for arrays with non-1 indices. For that reason, it is much better to avoid implementing these methods, and use the resulting to identify code that needs to be audited and perhaps generalized.

Do not annotate bounds checks

Julia 0.5 includes to annotate code that can be removed for callers that exploit . Initially, it seems far preferable to run with bounds checking always enabled (i.e., omit the annotation so the check always runs).

Custom types

If you're writing a non-1 indexed array type, you will want to specialize so it returns a , or (perhaps better) a custom . The advantage of a custom type is that it "signals" the allocation type for functions like . If we're writing an array type for which indexing will start at 0, we likely want to begin by creating a new , , where is equivalent to .

In general, you should probably *not* export from your package: there may be other packages that implement their own , and having multiple distinct types is (perhaps counterintuitively) an advantage: indicates that should create a , whereas indicates a type. This design allows peaceful coexistence among many different custom array types.

Note that the Julia package CustomUnitRanges.jl can sometimes be used to avoid the need to write your own type.

Specializing

Once you have your type, then specialize :

where here we imagine that has a field called (there would be other ways to implement this).

In some cases, the fallback definition for :

may not be what you want: you may need to specialize it to return something other than when . Likewise, in there is a dedicated function which is equivalent to but which avoids checking (at runtime) whether . (This is purely a performance optimization.) It is defined as:

If the first of these (the zero-dimensional case) is problematic for your custom array type, be sure to specialize it appropriately.

Specializing

Given your custom type, then you should also add the following two specializations for :

Both of these should allocate your custom array type.

Specializing

Optionally, define a method

and you can an array so that the result has custom indices.

Summary

Writing code that doesn't make assumptions about indexing requires a few extra abstractions, but hopefully the necessary changes are relatively straightforward.

As a reminder, this support is still experimental. While much of Julia's base code has been updated to support unconventional indexing, without a doubt there are many omissions that will be discovered only through usage. Moreover, at the time of this writing, most packages do not support unconventional indexing. As a consequence, early adopters should be prepared to identify and/or fix bugs. On the other hand, only through practical usage will it become clear whether this experimental feature should be retained in future versions of Julia; consequently, interested parties are encouraged to accept some ownership for putting it through its paces.

71.18 Base.LibGit2

The LibGit2 module provides bindings to libgit2, a portable C library that implements core functionality for the Git version control system. These bindings are currently used to power Julia's package manager. It is expected that this module will eventually be moved into a separate package.

Functionality

Some of this documentation assumes some prior knowledge of the libgit2 API. For more information on some of the objects and methods referenced here, consult the upstream libgit2 API reference.

– Type.

Abstract credentials payload

– Type.

A data buffer for exporting data from libgit2. Matches the struct.

When fetching data from LibGit2, a typical usage would look like:

In particular, note that should be called afterward on the object.

– Type.

Credentials that support caching

– Type.

Matches the struct.

– Type.

Matches the struct.

– Type.

Description of changes to one entry. Matches the struct.

The fields represent:

- : One of , indicating whether the file has been added/modified/deleted.
- : Flags for the delta and the objects on each side. Determines whether to treat the file(s) as binary/text, whether they exist on each side of the diff, and whether the object ids are known to be correct.

- : Used to indicate if a file has been renamed or copied.
- : The number of files in the delta (for instance, if the delta was run on a submodule commit id, it may contain more than one file).
- : A containing information about the file(s) before the changes.
- : A containing information about the file(s) after the changes.

– Type.

Description of one side of a delta. Matches the struct.

The fields represent:

- : the of the item in the diff. If the item is empty on this side of the diff (for instance, if the diff is of the removal of a file), this will be .
- : a terminated path to the item relative to the working directory of the repository.
- : the size of the item in bytes.
- : a combination of the flags. The th bit of this integer sets the th flag.
- : the mode for the item.
- : only present in LibGit2 versions newer than or equal to . The length of the field when converted using . Usually equal to (40).

– Type.

Matches the struct.

– Type.

Contains the information about HEAD during a fetch, including the name and URL of the branch fetched from, the oid of the HEAD, and whether the fetched HEAD has been merged locally.

– Type.

Matches the struct.

– Type.

Return a object from specified by /.

- is a full () or partial () hash.
- is a textual specification: see the git docs for a full list.

– Type.

Return a object from specified by /.

- is a full () or partial () hash.
- is a textual specification: see the git docs for a full list.

– Type.

A git object identifier, based on the sha-1 hash. It is a 20 byte string (40 hex digits) used to identify a in a repository.

– Type.

Return the specified object (, , or) from specified by /.

- is a full () or partial () hash.
- is a textual specification: see the git docs for a full list.

– Type.

Look up a remote git repository using its name and URL. Uses the default fetch refspec.

Examples

Look up a remote git repository using the repository's name and URL, as well as specifications for how to fetch from the remote (e.g. which remote branch to fetch from).

Examples

```
```

– Function.

```
```

Look up a remote git repository using only its URL, not its name.

Examples

```
```

– Type.

```
```

Open a git repository at .

– Function.

```
```

Open a git repository at with extended controls (for instance, if the current user must be a member of a special access group to read).

– Type.

```
```

A shortened git object identifier, which can be used to identify a git object when it is unique, consisting of the initial hexadecimal digits of (the remaining digits are ignored).

– Type.

```
```

This is a Julia wrapper around a pointer to a object.

– Type.

Collect information about the status of each file in the git repository (e.g. is the file modified, staged, etc.). can be used to set various options, for instance whether or not to look at untracked files or whether to include submodules or not.

– Type.

Return a object from specified by /.

- is a full () or partial () hash.
- is a textual specification: see the git docs for a full list.

– Type.

Return a object from specified by /.

- is a full () or partial () hash.
- is a textual specification: see the git docs for a full list.

– Type.

In-memory representation of a file entry in the index. Matches the struct.

– Type.

Matches the struct.

– Type.

Matches the struct.

– Type.

Options for connecting through a proxy.

Matches the struct.

The fields represent:

- : version of the struct in use, in case this changes later. For now, always .
- : an for the type of proxy to use. Defined in . The corresponding Julia enum is and has values:
  - : do not attempt the connection through a proxy.
  - : attempt to figure out the proxy configuration from the git configuration.
  - : connect using the URL given in the field of this struct.

  Default is to auto-detect the proxy type.
- : the URL of the proxy.
- : a pointer to a callback function which will be called if the remote requires authentication to connect.
- : a pointer to a callback function which will be called if certificate verification fails. This lets the user decide whether or not to keep connecting. If the function returns , connecting will be allowed. If it returns , the connection will not be allowed. A negative value can be used to return errors.
- : the payload to be provided to the two callback functions.

Examples

– Type.

Matches the struct.

The fields represent:

- : version of the struct in use, in case this changes later. For now, always .
- : if a pack file must be created, this variable sets the number of worker threads which will be spawned by the packbuilder. If , the packbuilder will auto-set the number of threads to use. The default is .
- : the callbacks (e.g. for authentication with the remote) to use for the push.
- : only relevant if the LibGit2 version is greater than or equal to . Sets options for using a proxy to communicate with a remote. See for more information.
- : only relevant if the LibGit2 version is greater than or equal to . Extra headers needed for the push operation.

– Type.

Describes a single instruction/operation to be performed during the rebase. Matches the struct.

– Type.

Matches the struct.

– Type.

Callback settings. Matches the struct.

– Type.

SSH credentials type

– Type.

An action signature (e.g. for committers, taggers, etc). Matches the struct.

– Type.

Providing the differences between the file as it exists in HEAD and the index, and providing the differences between the index and the working directory. Matches the struct.

– Type.

Options to control how will issue callbacks. Matches the struct.

– Type.

A LibGit2 representation of an array of strings. Matches the struct.

When fetching data from LibGit2, a typical usage would look like:

```

```

In particular, note that should be called afterward on the object.

Conversely, when passing a vector of strings to LibGit2, it is generally simplest to rely on implicit conversion:

```

```

Note that no call to is required as the data is allocated by Julia.

– Type.

```

```

Time in a signature. Matches the struct.

– Type.

Credentials that support only and parameters

– Function.

```

```

Add a *fetch* refspec for the specified . This refspec will contain information about which branch(es) to fetch from.

Examples

```

```

– Function.

```

```

Add a *push* refspec for the specified . This refspec will contain information about which branch(es) to push to.

Examples

```

```

Note
You may need to and reopen the in question after updating its push refspecs in order for the change to take effect and for calls to to work.

– Function.

Reads the file at and adds it to the object database of as a loose blob. Returns the of the resulting blob.

Examples

– Function.

Return the of the author of the commit . The author is the person who made changes to the relevant file(s). See also .

– Function.

Returns all authors of commits to the repository.

Examples

– Function.

Equivalent to . Create a new branch from the current HEAD.

– Function.

Checkout a new git branch in the repository. is the , in string form, which will be the start of the new branch. If is an empty string, the current HEAD will be used.

The keyword arguments are:

- : the name of the remote branch this new branch should track, if any. If empty (the default), no remote branch will be tracked.
- : if , branch creation will be forced.
- : if , after the branch creation finishes the branch head will be set as the HEAD of .

Equivalent to .

Examples

– Function.

Equivalent to . Checkout the git commit (a in string form) in . If is , force the checkout and discard any current changes. Note that this detaches the current HEAD.

Examples

– Function.

Checks if credentials were used

Checks if credentials were used or failed authentication, see

– Function.

Clone a remote repository located at to the local filesystem location .

The keyword arguments are:

- : which branch of the remote to clone, if not the default repository branch (usually).
- : if , clone the remote as a bare repository, which will make itself the git directory instead of . This means that a working tree cannot be checked out. Plays the role of the git CLI argument .
- : a callback which will be used to create the remote before it is cloned. If (the default), no attempt will be made to create the remote - it will be assumed to already exist.
- : provides credentials if necessary, for instance if the remote is a private repository.

Equivalent to .

Examples

– Function.

Wrapper around

Commit changes to repository

Commit the current patch to the rebase , using as the committer. Is silent if the commit has already been applied.

– Function.

Return the of the committer of the commit . The committer is the person who committed the changes originally authored by the , but need not be the same as the , for example, if the emailed a patch to a who committed it.

– Function.

Create a new branch in the repository with name , which points to commit (which has to be part of). If is , overwrite an existing branch named if it exists. If is and a branch already exists named , this function will throw an error.

– Function.

Credentials callback function

Function provides different credential acquisition functionality w.r.t. a connection protocol. If a payload is provided then should contain a object.

For type, if the payload contains fields: & , they are used to create authentication credentials. Empty name and word trigger an authentication error.

For type, if the payload contains fields: , , & , they are used to create authentication credentials. Empty name triggers an authentication error.

Credentials are checked in the following order (if supported):

- ssh key pair (if specified in payload's field)
- plain text

Note: Due to the specifics of the authentication procedure, when authentication fails, this function is called again without any indication whether authentication was successful or not. To avoid an infinite loop from repeatedly using the same faulty credentials, the function can be called. This function returns if the credentials were used. Using credentials triggers a user prompt for (re)entering required information. and are implemented using a call counting strategy that prevents repeated usage of faulty credentials.

– Function.

C function pointer for

– Function.

Return signature object. Free it after use.

– Function.

Delete the branch pointed to by .

– Function.

Show which files have changed in the git repository between branches and .

The keyword argument is:

- , and it sets options for the diff. The default is to show files added, modified, or deleted.

Returns only the *names* of the files which have changed, *not* their contents.

Examples

Equivalent to .

– Function.

Fetch from the specified remote git repository, using to determine which remote branch(es) to fetch. The keyword arguments are:

- : determines the options for the fetch, e.g. whether to prune afterwards.
- : a message to insert into the reflogs.

Fetches updates from an upstream of the repository .

The keyword arguments are:

- : which remote, specified by name, of to fetch from. If this is empty, the URL will be used to construct an anonymous remote.
- : the URL of . If not specified, will be assumed based on the given name of .
- : determines properties of the fetch.
- : provides credentials, if necessary, for instance if is a private repository.

Equivalent to .

– Function.

Get the *fetch* refspecs for the specified . These refspecs contain information about which branch(es) to fetch from.

Examples

– Function.

C function pointer for

– Method.

Perform a git merge on the repository , merging commits with diverging history into the current branch. Returns if the merge succeeded, if not.

The keyword arguments are:

- : Merge the named commit(s) in .
- : Merge the branch and all its commits since it diverged from the current branch.
- : If is , only merge if the merge is a fast-forward (the current branch head is an ancestor of the commits to be merged), otherwise refuse to merge and return . This is equivalent to the git CLI option .
- : specifies options for the merge, such as merge strategy in case of conflicts.
- : specifies options for the checkout step.

Equivalent to .

Note
If you specify a , this must be done in reference format, since the string will be turned into a . For example, if you wanted to merge branch , you would call .

– Function.

Fastforward merge changes into current head

– Function.

Return the name of the reference pointed to by the symbolic reference . If is not a symbolic reference, returns an empty string.

– Function.

Obtain the cached credentials for the given host+protocol (credid), or return and store the default if not found

– Function.

Return the location of the ”git” files of :

- for normal repositories, this is the location of the folder.
- for bare repositories, this is the location of the repository itself.

See also , .

– Function.

Returns a to the current HEAD of .

Return current HEAD of the repo as a string.

– Function.

Set the HEAD of to the object pointed to by .

– Function.

Lookup the object id of the current HEAD of git repository .

– Function.

Lookup the name of the current HEAD of git repository . If is currently detached, returns the name of the HEAD it’s detached from.

– Function.

Open a new git repository at . If is , the working tree will be created in . If is , no working directory will be created.

– Function.

Returns if , a in string form, is an ancestor of , a in string form.

Examples

– Function.

Use a heuristic to guess if a file is binary: searching for NULL bytes and looking for a reasonable ratio of printable to non-printable characters among the first 8000 bytes.

– Function.

Checks if commit (which is a in string form) is in the repository.

Examples

– Function.

Checks if there are any differences between the tree specified by and the tracked files in the working tree (if) or the index (if). are the specifications for options for the diff.

Examples

Equivalent to .

– Function.

Checks if there have been any changes to tracked files in the working tree (if) or the index (if). are the specifications for options for the diff.

Examples

Equivalent to .

– Function.

Checks if the current branch is an ”orphan” branch, i.e. has no commits. The first commit to this branch will have no parents.

– Function.

Determine if the branch specified by exists in the repository . If is , is assumed to be a remote git repository. Otherwise, it is part of the local filesystem.

returns a , which will be null if the requested branch does not exist yet. If the branch does exist, the contains a to the branch.

– Function.

Mirror callback function

Function sets refspecs and flag for remote reference.

– Function.

C function pointer for

– Function.

Return the commit message describing the changes made in commit . If is , return a slightly ”cleaned up” message (which has any leading newlines removed). If is , the message is not stripped of any such newlines.

– Function.

Return the full name of .

Get the name of a remote repository, for instance . If the remote is anonymous (see) the name will be an empty string .

Examples

The name of (e.g.).

– Function.

Equivalent to . Returns if needs updating.

– Function.

Returns the type corresponding to the enum value.

– Function.

Return the base file path of the repository .

- for normal repositories, this will typically be the parent directory of the ".git" directory (note: this may be different than the working directory, see for more details).
- for bare repositories, this is the location of the "git" files.

See also , .

– Function.

Recursively peel until an object of type is obtained. If no is provided, then will be peeled until an object other than a is obtained.

- A will be peeled to the object it references.
- A will be peeled to a .

Note
Only annotated tags can be peeled to objects. Lightweight tags (the default) are references under which point directly to objects.

Recursively peel until an object of type is obtained. If no is provided, then will be peeled until the type changes.

- A will be peeled to the object it references.
- A will be peeled to a .

– Function.

Standardise the path string to use POSIX separators.

– Function.

Push to the specified remote git repository, using to determine which remote branch(es) to push to. The keyword arguments are:

- : if , a force-push will occur, disregarding conflicts.
- : determines the options for the push, e.g. which proxy headers to use.

  Note

  You can add information about the push refspecs in two other ways: by setting an option in the repository's (with as the key) or by calling . Otherwise you will need to explicitly specify a push refspec in the call to for it to have any effect, like so: .

Pushes updates to an upstream of .

The keyword arguments are:

- : the name of the upstream remote to push to.
- : the URL of .
- : determines properties of the push.
- : determines if the push will be a force push, overwriting the remote branch.
- : provides credentials, if necessary, for instance if is a private repository.

Equivalent to .

– Function.

Get the *push* refspecs for the specified . These refspecs contain information about which branch(es) to push to.

Examples

– Function.

Read the tree (or the tree pointed to by in the repository owned by) into the index . The current index contents will be replaced.

– Function.

Attempt an automatic merge rebase of the current branch, from if provided, or otherwise from the upstream tracking branch. is the branch to rebase onto. By default this is .

If any conflicts arise which cannot be automatically resolved, the rebase will abort, leaving the repository and working tree in its original state, and the function will throw a . This is roughly equivalent to the following command line statement:

– Function.

Get a list of all reference names in the repository.

– Function.

Returns a corresponding to the type of :

- if the reference is invalid
- if the reference is an object id
- if the reference is symbolic

– Function.

Return a vector of the names of the remotes of .

– Function.

Resets credentials for another use

Updates some entries, determined by the , in the index from the target commit tree.

Sets the current head to the specified commit oid and optionally resets the index and working tree to match.

git reset [<committish>] [–] <pathspecs>...

Reset the repository to its state at , using one of three modes set by :

1. - move HEAD to .
2. - default, move HEAD to and reset the index to .
3. - move HEAD to , reset the index to , and discard all working changes.

Examples

In this example, the remote which is being fetched from *does* have a file called in its index, which is why we must reset.

Equivalent to .

Examples

– Function.

Return a repository to a previous , for example the HEAD of a branch before a merge attempt. can be generated using the function.

– Function.

List the number of revisions between and (committish OIDs in string form). Since and may be on different branches, performs a ”left-right” revision list (and count), returning a tuple of s - the number of left and right commits, respectively. A left (or right) commit refers to which side of a symmetric difference in a tree the commit is reachable from.

Equivalent to .

Examples

This will return .

– Function.

Set both the fetch and push for for the GitRepo or the git repository located at . Typically git repos use ”origin” as the remote name.

Examples

– Function.

Returns a shortened version of the name of that's "human-readable".

– Function.

Take a snapshot of the current state of the repository , storing the current HEAD, index, and any uncommitted work. The output can be used later during a call to to return the repository to the snapshotted state.

– Function.

Lookup the status of the file at in the git repository . For instance, this can be used to check if the file at has been modified and needs to be staged and committed.

– Function.

Get the stage number of . The stage number represents the current state of the working tree, but other numbers can be used in the case of a merge conflict. In such a case, the various stage numbers on an describe which side(s) of the conflict the current state of the file belongs to. Stage is the state before the attempted merge, stage is the changes which have been made locally, stages and larger are for changes from other branches (for instance, in the case of a multi-branch "octopus" merge, stages , , and might be used).

– Function.

Create a new git tag (e.g.) in the repository , at the commit .

The keyword arguments are:

- : the message for the tag.
- : if , existing references will be overwritten.
- : the tagger's signature.

– Function.

Remove the git tag from the repository .

– Function.

Get a list of all tags in the git repository .

– Function.

The of the target object of .

– Function.

Traverse the entries in a tree and its subtrees in post or pre order.

Function parameter should have following signature:

– Function.

Determine if the branch containing has a specified upstream branch.

returns a , which will be null if the requested branch does not have an upstream counterpart. If the upstream branch does exist, the contains a to the upstream branch.

– Function.

Get the fetch URL of a remote git repository.

Examples

– Function.

Resource management helper function

– Function.

Return the location of the working directory of . This will throw an error for bare repositories.

Note
This will typically be the parent directory of , but can be different in some cases: e.g. if either the configuration variable or the environment variable is set.

See also , .

71.19 Module loading

is responsible for loading modules and it also manages the precompilation cache. It is the implementation of the statement.

Experimental features

The features below are experimental and not part of the stable Julia API. Before building upon them inform yourself about the current thinking and whether they might change soon.

Module loading callbacks

It is possible to listen to the modules loaded by , by registering a callback.

Please note that the symbol given to the callback is a non-unique identifier and it is the responsibility of the callback provider to walk the module chain to determine the fully qualified name of the loaded binding.

The callback below is an example of how to do that:

71.20 Inference

How inference works

Type inference refers to the process of deducing the types of later values from the types of input values. Julia's approach to inference has been described in blog posts (1, 2).

Debugging inference.jl

You can start a Julia session, edit (for example to insert statements), and then replace in your running session by navigating to and executing . This trick typically leads to much faster development than if you rebuild Julia for each change.

A convenient entry point into inference is . Here's a demo running inference on :

If your debugging adventures require a , you can look it up by calling using many of the variables above. A object may be obtained with

The inlining algorithm (inline_worthy)

Much of the hardest work for inlining runs in . However, if your question is "why didn't my function inline?" then you will most likely be interested in and its primary callee, . handles a number of special cases (e.g., critical functions like and , incorporating a bonus for functions that return tuples, etc.). The main decision-making happens in , which returns if the function should be inlined.

implements a cost-model, where "cheap" functions get inlined; more specifically, we inline functions if their anticipated run-time is not large compared to the time it would take to issue a call to them if they were not inlined. The cost-model is extremely simple and ignores many important details: for example, all loops are analyzed as if they will be executed once, and the cost of an includes the summed cost of all branches. It's also worth acknowledging that we currently lack a suite of functions suitable for testing how well the cost model predicts the actual run-time cost, although BaseBenchmarks provides a great deal of indirect information about the successes and failures of any modification to the inlining algorithm.

The foundation of the cost-model is a lookup table, implemented in and its callers, that assigns an estimated cost (measured in CPU cycles) to each of Julia's intrinsic functions. These costs are based on standard ranges for common architectures (see Agner Fog's analysis for more detail).

We supplement this low-level lookup table with a number of special cases. For example, an expression (a call for which all input and output types were inferred in advance) is assigned a fixed cost (currently 20 cycles). In contrast, a expression, for functions other than intrinsics/builtins, indicates that the call will require dynamic dispatch, in which case we

assign a cost set by (currently set at 1000). Note that this is not a "first-principles" estimate of the raw cost of dynamic dispatch, but a mere heuristic indicating that dynamic dispatch is extremely expensive.

Each statement gets analyzed for its total cost in a function called . You can run this yourself by following this example:

The output is a holding the estimated cost of each statement in . Note that includes the consequences of inlining callees, and consequently the costs do too.

Chapter 72

Developing/debugging Julia's C code

72.1 Reporting and analyzing crashes (segfaults)

So you managed to break Julia. Congratulations! Collected here are some general procedures you can undergo for common symptoms encountered when something goes awry. Including the information from these debugging steps can greatly help the maintainers when tracking down a segfault or trying to figure out why your script is running slower than expected.

If you've been directed to this page, find the symptom that best matches what you're experiencing and follow the instructions to generate the debugging information requested. Table of symptoms:

- Segfaults during bootstrap ()
- Segfaults when running a script
- Errors during Julia startup

Version/Environment info

No matter the error, we will always need to know what version of Julia you are running. When Julia first starts up, a header is printed out with a version number and date. Please also include the output of in any report you create:

Segfaults during bootstrap ()

Segfaults toward the end of the process of building Julia are a common symptom of something going wrong while Julia is preparsing the corpus of code in the folder. Many factors can contribute toward this process dying unexpectedly,

however it is as often as not due to an error in the C-code portion of Julia, and as such must typically be debugged with a debug build inside of . Explicitly:

Create a debug build of Julia:

Note that this process will likely fail with the same error as a normal incantation, however this will create a debug executable that will offer the debugging symbols needed to get accurate backtraces. Next, manually run the bootstrap process inside of :

This will start , attempt to run the bootstrap process using the debug build of Julia, and print out a backtrace if (when) it segfaults. You may need to hit a few times to get the full backtrace. Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Segfaults when running a script

The procedure is very similar to Segfaults during bootstrap (). Create a debug build of Julia, and run your script inside of a debugged Julia process:

Note that will sit there, waiting for instructions. Type to run the process, and to generate a backtrace once it segfaults:

Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Errors during Julia startup

Occasionally errors occur during Julia's startup process (especially when using binary distributions, as opposed to compiling from source) such as the following:

These errors typically indicate something is not getting loaded properly very early on in the bootup phase, and our best bet in determining what's going wrong is to use external tools to audit the disk activity of the process:

- On Linux, use :
- On OSX, use :

Create a gist with the / ouput, the version info, and any other pertinent information and open a new issue on Github with a link to the gist.

Glossary

A few terms have been used as shorthand in this guide:

- refers to the root directory of the Julia source tree; e.g. it should contain folders such as , , , , etc.....

72.2 gdb debugging tips

Displaying Julia variables

Within , any object can be displayed using

The object will be displayed in the session, not in the gdb session. This is a useful way to discover the types and values of objects being manipulated by Julia's C code.

Similarly, if you're debugging some of Julia's internals (e.g.,), you can print using

This is a good way to circumvent problems that arise from the order in which julia's output streams are initialized.

Julia's flisp interpreter uses objects; these can be displayed with .

Useful Julia variables for Inspecting

While the addresses of many variables, like singletons, can be be useful to print for many failures, there are a number of additional variables (see for a complete list) that are even more useful.

- (when in) and :: for figuring out a bit about the call-stack
- and :: for figuring out what line in a test to go start debugging from (or figure out how far into a file has been parsed)
- :: not really a variable, but still a useful shorthand for referring to the result of the last gdb command (such as)
- :: sometimes useful, since it lists all of the command line options that were successfully parsed
- :: because who doesn't like to be able to interact with stdio

Useful Julia functions for Inspecting those variables

- :: For looking up the current function and line. (use on i686 platforms)
- :: For dumping the current Julia backtrace stack to stderr. Only usable after has been called.
- :: For invoking in gdb, where it doesn't work natively. For example, , , and .
- :: only works in lldb. Note: add something like to prevent lldb from printing its prompt over the output
- :: for invoking side-effects to modify the current state or to lookup symbols
- :: for extracting the type tag of a Julia value (in gdb, call first, or pick something short like for the first arg to define a shorthand)

Inserting breakpoints for inspection from gdb

In your session, set a breakpoint in like so:

Then within your Julia code, insert a call to by adding

where can be any variable or tuple you want to be accessible in the breakpoint.

It's particularly helpful to back up to the frame, from which you can display the arguments to a function using, e.g.,

Another useful frame is . The argument is a struct with a reference to the final AST sent into the compiler. However, the AST at this point will usually be compressed; to view the AST, call and then pass the result to :

Inserting breakpoints upon certain conditions

Loading a particular file

Let's say the file is :

Calling a particular method

Since this function is used for every call, you will make everything 1000x slower if you do this.

Dealing with signals

Julia requires a few signal to function property. The profiler uses for sampling and the garbage collector uses for threads synchronization. If you are debugging some code that uses the profiler or multiple threads, you may want to let the debugger ignore these signals since they can be triggered very often during normal operations. The command to do this in GDB is (replace with or other signals you want to ignore):

The corresponding LLDB command is (after the process is started):

If you are debugging a segfault with threaded code, you can set a breakpoint on (should also work on Linux and BSD) in order to only catch the actual segfault rather than the GC synchronization points.

Debugging during Julia's build process (bootstrap)

Errors that occur during need special handling. Julia is built in two stages, constructing and . To see what commands are running at the time of failure, use .

At the time of this writing, you can debug build errors during the phase from the directory using:

You might need to delete all the files in to get this to work.

You can debug the phase using:

By default, any errors will cause Julia to exit, even under gdb. To catch an error "in the act", set a breakpoint in (there are several other useful spots, for specific kinds of failures, including: , , and).

Once an error is caught, a useful technique is to walk up the stack and examine the function by inspecting the related call to . To take a real-world example:

The most recent is at frame #3, so we can go back there and look at the AST for the function . This is the uniqued name for some method of . in this frame is a , so we can look at the type signature, if any, from the field:

Then, we can look at the AST for this function:

Finally, and perhaps most usefully, we can force the function to be recompiled in order to step through the codegen process. To do this, clear the cached from the :

Then, set a breakpoint somewhere useful (e.g. , , , etc.), and run codegen:

Debugging precompilation errors

Module precompilation spawns a separate Julia process to precompile each module. Setting a breakpoint or catching failures in a precompile worker requires attaching a debugger to the worker. The easiest approach is to set the debugger watch for new process launches matching a given name. For example:

or:

Then run a script/command to start precompilation. As described earlier, use conditional breakpoints in the parent process to catch specific file-loading events and narrow the debugging window. (some operating systems may require alternative approaches, such as following each from the parent process)

Mozilla's Record and Replay Framework (rr)

Julia now works out of the box with rr, the lightweight recording and deterministic debugging framework from Mozilla. This allows you to replay the trace of an execution deterministically. The replayed execution's address spaces, register contents, syscall data etc are exactly the same in every run.

A recent version of rr (3.1.0 or higher) is required.

72.3 Using Valgrind with Julia

Valgrind is a tool for memory debugging, memory leak detection, and profiling. This section describes things to keep in mind when using Valgrind to debug memory issues with Julia.

General considerations

By default, Valgrind assumes that there is no self modifying code in the programs it runs. This assumption works fine in most instances but fails miserably for a just-in-time compiler like . For this reason it is crucial to pass to , else code may crash or behave unexpectedly (often in subtle ways).

In some cases, to better detect memory errors using Valgrind it can help to compile with memory pools disabled. The compile-time flag disables memory pools in Julia, and disables memory pools in FemtoLisp. To build with both flags, add the following line to :

Another thing to note: if your program uses multiple workers processes, it is likely that you want all such worker processes to run under Valgrind, not just the parent process. To do this, pass to .

Suppressions

Valgrind will typically display spurious warnings as it runs. To reduce the number of such warnings, it helps to provide a suppressions file to Valgrind. A sample suppressions file is included in the Julia source distribution at .

The suppressions file can be used from the source directory as follows:

Any memory errors that are displayed should either be reported as bugs or contributed as additional suppressions. Note that some versions of Valgrind are shipped with insufficient default suppressions, so that may be one thing to consider before submitting any bugs.

Running the Julia test suite under Valgrind

It is possible to run the entire Julia test suite under Valgrind, but it does take quite some time (typically several hours). To do so, run the following command from the directory:

If you would like to see a report of "definite" memory leaks, pass the flags to as well.

Caveats

Valgrind currently does not support multiple rounding modes, so code that adjusts the rounding mode will behave differently when run under Valgrind.

In general, if after setting you find that your program behaves differently when run under Valgrind, it may help to pass to as you investigate further. This will enable the minimal Valgrind machinery but will also run much faster than when the full memory checker is enabled.

72.4 Sanitizer support

General considerations

Using Clang's sanitizers obviously require you to use Clang (), but there's another catch: most sanitizers require a runtime library, provided by the host compiler, while the instrumented code generated by Julia's JIT relies on functionality from that library. This implies that the LLVM version of your host compiler matches that of the LLVM library used within Julia.

An easy solution is to have an dedicated build folder for providing a matching toolchain, by building with . You can then refer to this toolchain from another build folder by specifying while overriding the and variables.

Address Sanitizer (ASAN)

For detecting or debugging memory bugs, you can use Clang's address sanitizer (ASAN). By compiling with you enable ASAN for the Julia compiler and its generated code. In addition, you can specify to sanitize the LLVM library as well. Note that these options incur a high performance and memory cost. For example, using ASAN for Julia and LLVM makes takes 8-10 times as long while using 20 times as much memory (this can be reduced to respectively a factor of 3 and 4 by using the options described below).

By default, Julia sets the ASAN flag, which is required for signal delivery to work properly. You can define other options using the environment flag, in which case you'll need to repeat the default option mentioned before. For example, memory usage can be reduced by specifying and , at the cost of backtrace accuracy. For now, Julia also sets , but this should be removed in the future.

Memory Sanitizer (MSAN)

For detecting use of uninitialized memory, you can use Clang's memory sanitizer (MSAN) by compiling with .