Julia: a programming language for scientific computing

Bill McLean
School of Maths & Stats, UNSW

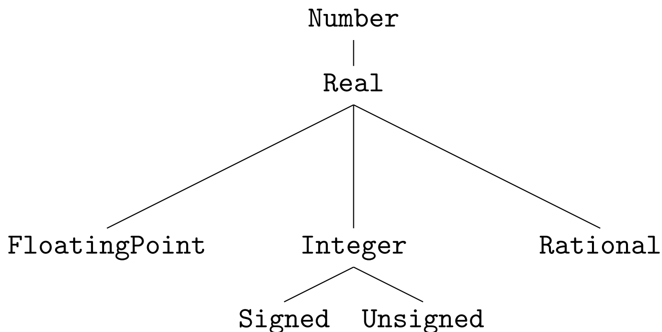
Version: January 13, 2014

Part I

A first look at Julia

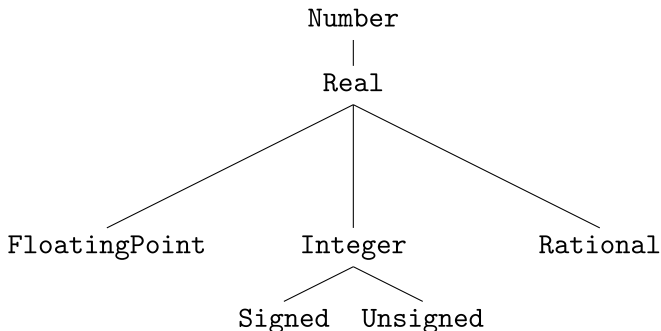
Julia's real data types

Hierarchy of types:



Julia's real data types

Hierarchy of types:



Concrete `FloatingPoint` types (bits types):

`Float16`, `Float32`, `Float64`, `BigFloat`.

Functions

Simple example:

```
function polar(x, y)
    # Returns polar coordinates
    r = hypot(x, y)
    theta = atan2(y, x)
    return r, theta
end
```

Functions

Simple example:

```
function polar(x, y)
    # Returns polar coordinates
    r = hypot(x, y)
    theta = atan2(y, x)
    return r, theta
end
```

Default values and keyword arguments:

```
function succ(x, step=1; count=1)
    x + count * step
end
```

Functions

Simple example:

```
function polar(x, y)
    # Returns polar coordinates
    r = hypot(x, y)
    theta = atan2(y, x)
    return r, theta
end
```

Default values and keyword arguments:

```
function succ(x, step=1; count=1)
    x + count * step
end
```

Many other features.

Arrays

Indexed from 1 but elements accessed using square brackets, e.g., if

```
A = [ 1  -5  3
      2   4  0 ]
```

then `A[2,1]` equals 2.

Arrays

Indexed from 1 but elements accessed using square brackets, e.g., if

```
A = [ 1  -5  3
      2   4  0 ]
```

then `A[2,1]` equals 2.

Entries stored in Fortran (column-major) order.

Arrays

Indexed from 1 but elements accessed using square brackets, e.g., if

```
A = [ 1  -5  3
      2   4  0 ]
```

then `A[2,1]` equals 2.

Entries stored in Fortran (column-major) order.

Index ranges `start:stride:finish` same as in Matlab (with `stride` equal to 1 by default).

Arrays

Indexed from 1 but elements accessed using square brackets, e.g., if

```
A = [ 1  -5  3
      2   4  0 ]
```

then `A[2,1]` equals 2.

Entries stored in Fortran (column-major) order.

Index ranges `start:stride:finish` same as in Matlab (with `stride` equal to 1 by default).

Can construct arrays using comprehensions

```
A = [ exp(x[i]) * sin(y[j]) for i=1:m, j=1:n ]
```

Modules

A large application can be split into modules, each with its own namespace.

```
module GaussQuadrature

export legendre, jacobi, laguerre, hermite

function legendre(n)
    ...
end

end
```

Modules

A large application can be split into modules, each with its own namespace.

```
module GaussQuadrature

export legendre, jacobi, laguerre, hermite

function legendre(n)
    ...
end

end
```

Several ways to access functions and data in a module.

```
using GaussQuadrature
x, w = legendre(5)
```

Standard library

Easy access to Lapack/BLAS, FFTW, SuiteSparse.

Standard library

Easy access to Lapack/BLAS, FFTW, SuiteSparse.

Otherwise, the standard library is implemented in Julia.

Standard library

Easy access to Lapack/BLAS, FFTW, SuiteSparse.

Otherwise, the standard library is implemented in Julia.

Basic support for statistics, sorting, text processing, dictionaries, quadrature.

Standard library

Easy access to Lapack/BLAS, FFTW, SuiteSparse.

Otherwise, the standard library is implemented in Julia.

Basic support for statistics, sorting, text processing, dictionaries, quadrature.

Numerous third-party packages add splines, ODE solvers, optimization, data handling and graphics.

Is that legal?

Many common student errors are accepted by Julia.

- `y = e^x  # x is a scalar`

Is that legal?

Many common student errors are accepted by Julia.

- `y = e^x  # x is a scalar`
- `y = 7x - 2  # no space after 7`

Is that legal?

Many common student errors are accepted by Julia.

- `y = e^x  # x is a scalar`
- `y = 7x - 2  # no space after 7`
- `f(x) = (x-1)cos(pi*x) # x is a scalar`

Is that legal?

Many common student errors are accepted by Julia.

- `y = e^x  # x is a scalar`
- `y = 7x - 2  # no space after 7`
- `f(x) = (x-1)cos(pi*x) # x is a scalar`

But strict treatment of argument types.

- `log(-1)` raises a `DomainError`.

Is that legal?

Many common student errors are accepted by Julia.

- `y = e^x  # x is a scalar`
- `y = 7x - 2  # no space after 7`
- `f(x) = (x-1)cos(pi*x) # x is a scalar`

But strict treatment of argument types.

- `log(-1)` raises a `DomainError`.
- `log(-1+0im)` returns $i\pi$.

Part II

Distinguishing features

History

- `www.julialang.org`

History

- `www.julialang.org`
- Development begun in 2009 at MIT. Lead developers Alan Edelman, Jeff Bezanson, Stefan Karpinski, Viral Shah.

History

- `www.julialang.org`
- Development begun in 2009 at MIT. Lead developers Alan Edelman, Jeff Bezanson, Stefan Karpinski, Viral Shah.
- Liberal MIT licence.

History

- `www.julialang.org`
- Development begun in 2009 at MIT. Lead developers Alan Edelman, Jeff Bezanson, Stefan Karpinski, Viral Shah.
- Liberal MIT licence.
- Initial public announcement February 2012.
- Current (November 2013) version 0.2.

History

- www.julialang.org
- Development begun in 2009 at MIT. Lead developers Alan Edelman, Jeff Bezanson, Stefan Karpinski, Viral Shah.
- Liberal MIT licence.
- Initial public announcement February 2012.
- Current (November 2013) version 0.2.
- Already a substantial user base with active mailing lists etc.

LLVM

Julia uses a Just-in-Time (JIT) compiler built with the Low-Level Virtual Machine (LLVM).

LLVM

Julia uses a Just-in-Time (JIT) compiler built with the Low-Level Virtual Machine (LLVM).

Example

```
for j=1:N, i=1:N
    a[i,j] = complicated expression in x[i] and x[j]
end
```

On my laptop (Core i5-3360M @ 2.80GHz), with $N = 2000$:

ifort	0.11s
julia	0.28s
gfortran	0.39s
matlab	0.52s
octave	115.60s

Don't vectorize for performance

Example

Comparison with

```
for j=1:N
    a[:,j] = complicated expression in x[:] and x[j]
end
```

shows

	vectorized	nested loops
julia	0.40s	0.28s
matlab	0.23s	0.52s
octave	0.45s	115.60s

Type assertions

The function call `sine_approx(n,x)` fails unless `n` is a subtype of `Integer` and `x` is a subtype of `Number`.

```
function sine_approx(n::Integer, x::Number)
    # Taylor approximation to sin(x) of degree 2n-1.
    s = t = one(x); xsqr = x^2
    for k = 1:n-1
        t = - t * xsqr / ( (2k+1)*(2k) )
        s += t
    end
    return x * s
end
```

Generic functions

Julia can generate machine code for multiple versions of a function, one for each permitted sequence of concrete argument types.

```
code_native(sine_approx, (Int32, Float32))
code_native(sine_approx, (Int64, Float64))
code_native(sine_approx, (Int64, BigFloat))
code_native(sine_approx, (Int64, Complex{Float64}))
```

Generic functions

Julia can generate machine code for multiple versions of a function, one for each permitted sequence of concrete argument types.

```
code_native(sine_approx, (Int32, Float32))
code_native(sine_approx, (Int64, Float64))
code_native(sine_approx, (Int64, BigFloat))
code_native(sine_approx, (Int64, Complex{Float64}))
```

In practice, Julia compiles only the versions that are actually called in a given program.

The programmer can also define specific methods to optimize for particular argument types.

Calling C/Fortran

Suppose the shared library `sine_approx.so` holds a double precision Fortran version of our function. We create a Julia wrapper as follows.

```
function sine_approx(n::Int64, x::Float64)
    y = ccall( (:sine_approx_, "./sine_approx.so"),
              Float64,
              (Ptr{Int64}, Ptr{Float64}),
              &n, &x)
    return y
end
```

Now, `sine_approx(n,x)` calls the Fortran version whenever the Julia types of `n` and `x` are `Int64` and `Float64`.

Multiple dispatch

The command

```
methods(sine_approx)
```

lists the available implementations (methods) of the `sine_approx` function:

```
# methods for generic function sine_approx
sine_approx(n::Int64,x::Float64) at sine_approx.jl:20
sine_approx(n::Integer,x::Number) at sine_approx.jl:3
sine_approx(n::Integer,x::Array{T,N}) at sine_approx.jl:12
```

Multiple dispatch

The command

```
methods(sine_approx)
```

lists the available implementations (methods) of the sine_approx function:

```
# methods for generic function sine_approx
sine_approx(n::Int64,x::Float64) at sine_approx.jl:20
sine_approx(n::Integer,x::Number) at sine_approx.jl:3
sine_approx(n::Integer,x::Array{T,N}) at sine_approx.jl:12
```

Each time a function is called, Julia chooses the most specific method based on the types of all of the actual arguments.

```
sine_approx(3,0.3-0.5im) # (Integer, Number)
sine_approx(4, 0.76)     # (Int64, Float64)
sine_approx(5, [0.0, 0.2]) # (Integer, Array{Float64,1})
```

Method parameters

```
module GaussQuadrature

function jacobi{T<:FloatingPoint}(n::Integer,
         alpha::T, beta::T, endpt::EndPt=neither)
    @assert alpha > -one(T) && beta > -one(T)
    a, b, muzero = jacobi_coeff(n, alpha, beta, endpt)
    custom_gauss_rule(-one(T), one(T), a, b,
                      muzero, endpt)
end
```

Method parameters

```
module GaussQuadrature

function jacobi{T<:FloatingPoint}(n::Integer,
         alpha::T, beta::T, endpt::EndPt=neither)
    @assert alpha > -one(T) && beta > -one(T)
    a, b, muzero = jacobi_coeff(n, alpha, beta, endpt)
    custom_gauss_rule(-one(T), one(T), a, b,
                      muzero, endpt)
end
```

In a function call

```
x, w = jacobi(n, alpha, beta)
```

the arguments `alpha` and `beta` must be of the same type, which must be a subtype of `FloatingPoint`.

Parallel execution

- Julia implements a distributed-memory, parallel execution model based on one-sided message passing.

Parallel execution

- Julia implements a distributed-memory, parallel execution model based on one-sided message passing.
- Maybe shared memory parallel computing in future, since recent support for OpenMP in LLVM.

Parallel execution

- Julia implements a distributed-memory, parallel execution model based on one-sided message passing.
- Maybe shared memory parallel computing in future, since recent support for OpenMP in LLVM.
- Current implementation includes some easy high-level constructs.

Parallel execution

- Julia implements a distributed-memory, parallel execution model based on one-sided message passing.
- Maybe shared memory parallel computing in future, since recent support for OpenMP in LLVM.
- Current implementation includes some easy high-level constructs.
- Standard library includes threaded Lapack/Blas (OpenBlas).

Parallel execution

Start Julia with 3 processes.

```
$ julia -p 2
```

Parallel execution

Start Julia with 3 processes.

```
$ julia -p 2
```

At the Julia prompt.

```
julia> nprocs(), nworkers()
(3,2)

julia> require("matrix.jl")

julia> @time a1=matrix(5000);
elapsed time: 1.663932175 seconds (200040144 bytes allocate

julia> @time a=pmap(matrix, [5000,5001]);
elapsed time: 2.275647082 seconds (400406784 bytes allocate
```

Part III

Scientific programming languages

The tower of Babel

1950s	Fortran
1960s	Algol, Macsyma
1970s	Pascal, C, IDL
1980s	C++, Maple, Matlab, Mathematica
1990s	Scilab, Octave, Python, R
2000s	FreeMat, NumPy/SciPy/SymPy, Sage
2010s	Julia

The tower of Babel

1950s	Fortran
1960s	Algol, Macsyma
1970s	Pascal, C, IDL
1980s	C++, Maple, Matlab, Mathematica
1990s	Scilab, Octave, Python, R
2000s	FreeMat, NumPy/SciPy/SymPy, Sage
2010s	Julia

The TIOBE Programming Community Index for November 2013 lists C, C++, Python in the top 10, and Matlab at 16th.

Mixed language programming

Current paradigm: rapidly develop application code in a convenient scripting language (Python) complemented by compiled (C/Fortran) libraries of fast computational routines, plus graphics.

Mixed language programming

Current paradigm: rapidly develop application code in a convenient scripting language (Python) complemented by compiled (C/Fortran) libraries of fast computational routines, plus graphics.

- No problem if reuse of existing libraries suffices.

Mixed language programming

Current paradigm: rapidly develop application code in a convenient scripting language (Python) complemented by compiled (C/Fortran) libraries of fast computational routines, plus graphics.

- No problem if reuse of existing libraries suffices.
- Writing custom libraries requires higher level of programmer knowledge/skill.

Mixed language programming

Current paradigm: rapidly develop application code in a convenient scripting language (Python) complemented by compiled (C/Fortran) libraries of fast computational routines, plus graphics.

- No problem if reuse of existing libraries suffices.
- Writing custom libraries requires higher level of programmer knowledge/skill.
- Many tools: ctypes, f2py, SWIG, Cython.

Mixed language programming

Current paradigm: rapidly develop application code in a convenient scripting language (Python) complemented by compiled (C/Fortran) libraries of fast computational routines, plus graphics.

- No problem if reuse of existing libraries suffices.
- Writing custom libraries requires higher level of programmer knowledge/skill.
- Many tools: ctypes, f2py, SWIG, Cython.
- Is JIT compiler (Matlab, PyPy, Julia) fast enough?

Python, Octave	99% of UG problems
Matlab, Julia	95% of research problems
C, Fortran (parallel)	Serious HPC applications

Other tradeoffs in language choice

- Licencing: proprietory vs free software.

Other tradeoffs in language choice

- Licencing: proprietory vs free software.
- General purpose (C, Python) vs domain specific (Fortran, Matlab, Julia).

Other tradeoffs in language choice

- Licencing: proprietory vs free software.
- General purpose (C, Python) vs domain specific (Fortran, Matlab, Julia).
- Business model: Mathworks vs Enthought.

Other tradeoffs in language choice

- Licencing: proprietory vs free software.
- General purpose (C, Python) vs domain specific (Fortran, Matlab, Julia).
- Business model: Mathworks vs Enthought.
- Officially supported vs third party packages.

Other tradeoffs in language choice

- Licencing: proprietory vs free software.
- General purpose (C, Python) vs domain specific (Fortran, Matlab, Julia).
- Business model: Mathworks vs Enthought.
- Officially supported vs third party packages.
- Documentation.

Other tradeoffs in language choice

- Licencing: proprietory vs free software.
- General purpose (C, Python) vs domain specific (Fortran, Matlab, Julia).
- Business model: Mathworks vs Enthought.
- Officially supported vs third party packages.
- Documentation.
- User base.

Other tradeoffs in language choice

- Licencing: proprietory vs free software.
- General purpose (C, Python) vs domain specific (Fortran, Matlab, Julia).
- Business model: Mathworks vs Enthought.
- Officially supported vs third party packages.
- Documentation.
- User base.
- Longevity (past and future), standardisation.

Other tradeoffs in language choice

- Licencing: proprietory vs free software.
- General purpose (C, Python) vs domain specific (Fortran, Matlab, Julia).
- Business model: Mathworks vs Enthought.
- Officially supported vs third party packages.
- Documentation.
- User base.
- Longevity (past and future), standardisation.
- Stand alone environment vs Unix-friendly compiler/interpreter.

Teaching

- Institutional inertia: do we need automatic review every n years?

Teaching

- Institutional inertia: do we need automatic review every *n* years?
- Obvious advantage to common progamming language across courses, but change is then more difficult.

Teaching

- Institutional inertia: do we need automatic review every n years?
- Obvious advantage to common progamming language across courses, but change is then more difficult.
- Service teaching: the customer is always right.

Teaching

- Institutional inertia: do we need automatic review every n years?
- Obvious advantage to common progamming language across courses, but change is then more difficult.
- Service teaching: the customer is always right.
- How much does a choice of language matter?

Teaching

- Institutional inertia: do we need automatic review every n years?
- Obvious advantage to common progamming language across courses, but change is then more difficult.
- Service teaching: the customer is always right.
- How much does a choice of language matter?
- "I've seen the best minds of my generation destroyed by Matlab . . . ".