Overview

Julia is a language designed for technical computing.

Avoids the two language problem

- Scripting ease of a dynamic language
- Just-In-Time compiled performance like a static language
  - Scalar code runs fast
  - Vectorizer
  - Parallelism
    - Distributed memory via one-sided communication
    - Shared memory is a work in progress

Active open-source community

- 400 contributors
Outline

Brief history of Julia
Taste of the language
Compilation technology
## Brief History

<table>
<thead>
<tr>
<th>Year</th>
<th>Event/Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>started</td>
</tr>
<tr>
<td>2012</td>
<td>v. 0.1 released. &quot;Because we are greedy.&quot;</td>
</tr>
<tr>
<td>2013</td>
<td>v. 0.2</td>
</tr>
<tr>
<td></td>
<td>3rd party IDE released</td>
</tr>
<tr>
<td>2014</td>
<td>JuliaCon 2014</td>
</tr>
<tr>
<td></td>
<td>v. 0.3</td>
</tr>
<tr>
<td></td>
<td>JuliaBox</td>
</tr>
<tr>
<td>2015</td>
<td>Issue #8839 closed.</td>
</tr>
<tr>
<td></td>
<td>JuliaCon 2015</td>
</tr>
<tr>
<td></td>
<td><em>Tupocaplypse</em></td>
</tr>
<tr>
<td></td>
<td>v. 0.4?</td>
</tr>
<tr>
<td></td>
<td><em>Arraymageddon</em></td>
</tr>
<tr>
<td>Year</td>
<td>Event</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>2009</td>
<td>started</td>
</tr>
<tr>
<td>2012</td>
<td>v. 0.1 released, “Because we are greedy.”</td>
</tr>
<tr>
<td>2013</td>
<td>v. 0.2, 3rd party IDE released</td>
</tr>
<tr>
<td>2014</td>
<td>JuliaCon 2014, v. 0.3, JuliaBox</td>
</tr>
<tr>
<td>2015</td>
<td>Issue #8839 closed. JuliaCon 2015, Tupocaplypse, v. 0.4? Arraymageddon</td>
</tr>
</tbody>
</table>
Brief History

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>started</td>
</tr>
</tbody>
</table>
| 2012 | v. 0.1 released  
“Because we are greedy.” |
| 2013 | v. 0.2  
3rd party IDE released |
| 2014 | JuliaCon 2014  
v. 0.3  
JuliaBox |
| 2015 | Issue #8839 closed.  
JuliaCon 2015  
Tupocaplypse  
v. 0.4?  
Arraymageddon |

Personal history

“Because we are greedy.”

Googled “language for technical computing”

Intel® MKL build option

Intel® VTune support
Brief History

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>Started</td>
</tr>
<tr>
<td>2012</td>
<td>v. 0.1 released &quot;Because we are greedy.&quot;</td>
</tr>
<tr>
<td>2013</td>
<td>v. 0.2 3rd party IDE released</td>
</tr>
<tr>
<td>2014</td>
<td>JuliaCon 2014 v. 0.3 JuliaBox</td>
</tr>
<tr>
<td>2015</td>
<td>Issue #8839 closed. JuliaCon 2015 Tupocaplypse v. 0.4? Arraymageddon</td>
</tr>
</tbody>
</table>

Personal history

Googled “language for technical computing”
Intel® MKL build option
Intel® VTune support
Type-based alias analysis
Vectorization
### Brief History

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>started</td>
</tr>
<tr>
<td>2012</td>
<td>v. 0.1 released “Because we are greedy.”</td>
</tr>
<tr>
<td>2013</td>
<td>v. 0.2 3rd party IDE released</td>
</tr>
<tr>
<td>2014</td>
<td>JuliaCon 2014 v. 0.3 JuliaBox</td>
</tr>
<tr>
<td>2015</td>
<td>Issue #8839 closed. JuliaCon 2015 Tupocaplypse v. 0.4? Arraymageddon</td>
</tr>
</tbody>
</table>

### Personal history

- Googled “language for technical computing”
- Intel® MKL build option
- Intel® VTune support
- Type-based alias analysis
- Vectorization
- 2nd in Al Zimmerman contest
- JuliaCon workshop
- Work on threading branch
“Why We Created Julia”
14 Feb 2012 | Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman

In short, because we are greedy.

...

We want a language that’s open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that’s homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

(Did we mention it should be as fast as C?)

...
Taste of Julia language
Creature Comforts

REPL with LaTeX auto-complete
Built-in help documentation
Built-in package manager
IJulia environment
Julia Box: https://www.juliabox.org/
Julia Example

function delacorte\{T<:Integer\}(a::Matrix\{T\})
    (m,n) = size(a)
    s = 0
    for x0=1:n, x1=1:n, y0=1:m, y1=1:m
        s += gcd(a[y0,x0],a[y1,x1]) *
            ((x0-x1)^2 + (y0-y1)^2)
    end
    @assert iseven(s)
    return div(s,2)
end

blue = keyword
brown = standard library
Another Example

```plaintext
function axpy( a, x, y )
    @simd for i=1:length(x)
        @inbounds y[i] += a*x[i]
    end
end
```

Must turn off bounds-checking for vectorization to work.

blue = keyword
brown = standard library
Multiple Dispatch

Excerpt from Julia standard library

```julia
immutable Complex{T<:Real} <: Number
    re::T
    im::T
end

*(z::Complex, w::Complex) = Complex(real(z) * real(w) - imag(z) * imag(w),
    real(z) * imag(w) + imag(z) * real(w))

*(x::Real, z::Complex) = Complex(x * real(z), x * imag(z))
*(z::Complex, x::Real) = Complex(x * real(z), x * imag(z))
```
Turtles All the Way Down

Excerpted from Julia standard library

bitstype 32 Float32 <: AbstractFloat
bitstype 64 Float64 <: AbstractFloat

blue = keyword
brown = predefined
Turtles All the Way Down

Excerpted from Julia standard library

```plaintext
bitstype 32 Float32 <: AbstractFloat
bitstype 64 Float64 <: AbstractFloat

*(x::Float32, y::Float32) = box(Float32,mul_float(unbox(Float32,x), unbox(Float32,y)))

*(x::Float64, y::Float64) = box(Float64,mul_float(unbox(Float64,x), unbox(Float64,y)))
```

blue = keyword
brown = predefined
red = intrinsic
Can Call C Efficiently

Don’t need the C source.
Can call any function in a shared library that follows C ABI.
Can Call C Efficiently

foo(x,y) = ccall((:atan2f,"libm"), Float32, (Float32,Float32), x, y) / 2\pi

Don't need the C source.
Can call any function in a shared library that follows C ABI.
Can Call C Efficiently

Don't need the C source.
Can call any function in a shared library that follows C ABI.

foo(x,y) = \texttt{ccall}((\texttt{atan2f},"libm"), \texttt{Float32}, (\texttt{Float32,Float32}), x, y) / 2\pi
Can Call C Efficiently

foo(x,y) = \texttt{ccall}(:atan2f,"libm"), \texttt{Float32}, (\texttt{Float32}, \texttt{Float32}), x, y) / 2\pi

Don't need the C source.
Can call any function in a shared library that follows C ABI.
Can Call C Efficiently

foo(x,y) = \texttt{ccall}((\texttt{:atan2f,"libm"}), \texttt{Float32}, (\texttt{Float32,Float32}), x, y) / 2\pi

Don't need the C source.
Can call any function in a shared library that follows C ABI.
Can Call C Efficiently

\[ \text{foo}(x, y) = \text{ccall}((:\text{atan2f},"libm"), \text{Float32}, (\text{Float32, Float32}), x, y) / 2\pi \]

Don't need the C source.
Can call any function in a shared library that follows C ABI.
Can Call C Efficiently

foo(x,y) = \texttt{ccall}((:atan2f,"libm"), \texttt{Float32}, (\texttt{Float32},\texttt{Float32}), x, y) / 2\pi

Code generated for foo when called with Float64 arguments.

Don't need the C source.
Can call any function in a shared library that follows C ABI.
Metaprogramming
Metaprogramming

Excerpted from Julia standard library

```julia
    @eval begin
        ($f)(x::Float64) = ccall((string(f), libm), Float64, (Float64,), x)
        ($f)(x::Float32) = ccall((string(f, "f"), libm), Float32, (Float32,), x)
        ($f)(x::Real) = ($f)(float(x))
        @vectorize_1arg Number $f
    end
end
```
Metaprogramming

Excerpted from Julia standard library

```julia
    @eval begin
        ($f)(x::Float64) = ccall((string(f)),libm), Float64, (Float64,), x)
        ($f)(x::Float32) = ccall((string(f,"f")),libm), Float32, (Float32,), x)
        ($f)(x::Real) = ($f)(float(x))
    end
@vectorize_1arg Number $f
end
```

Create default for Integer, Irrational, Rational, Float16.
Metaprogramming

Excerpted from Julia standard library

```julia
    @eval begin
        ($f)(x::Float64) = ccall(($string(f)),libm), Float64, (Float64,), x)
        ($f)(x::Float32) = ccall(($string(f, "f")),libm), Float32, (Float32,), x)
        ($f)(x::Real) = ($f)(float(x))
    end
end

Create overloads for array arguments.
Create default for Integer, Irrational, Rational, Float16.
```
## Performance Relative to C

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>Julia</th>
<th>Python</th>
<th>R</th>
<th>Matlab</th>
<th>Octave</th>
<th>Mathematica</th>
<th>JavaScript</th>
<th>Go</th>
<th>LuaJIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcc 4.8.2</td>
<td>0.3.7</td>
<td>2.7.9</td>
<td>3.1.3</td>
<td>R2014a</td>
<td>3.8.1</td>
<td>10.0</td>
<td>V8 3.14.5.9</td>
<td>go1.2.1</td>
<td>gsl-shell 2.3.1</td>
</tr>
<tr>
<td><strong>fib</strong></td>
<td>0.57</td>
<td>2.14</td>
<td>95.45</td>
<td>528.85</td>
<td>4258.12</td>
<td>9211.59</td>
<td>166.64</td>
<td>3.68</td>
<td>2.20</td>
<td>2.02</td>
</tr>
<tr>
<td><strong>parse_int</strong></td>
<td>4.67</td>
<td>1.57</td>
<td>20.48</td>
<td>54.30</td>
<td>1525.88</td>
<td>7568.38</td>
<td>17.70</td>
<td>2.29</td>
<td>3.78</td>
<td>6.09</td>
</tr>
<tr>
<td><strong>quicksort</strong></td>
<td>1.10</td>
<td>1.21</td>
<td>46.70</td>
<td>248.28</td>
<td>55.87</td>
<td>1532.54</td>
<td>48.47</td>
<td>2.91</td>
<td>1.09</td>
<td>2.00</td>
</tr>
<tr>
<td><strong>mandel</strong></td>
<td>0.87</td>
<td>0.87</td>
<td>18.83</td>
<td>58.97</td>
<td>60.09</td>
<td>393.91</td>
<td>6.12</td>
<td>1.86</td>
<td>1.17</td>
<td>0.71</td>
</tr>
<tr>
<td><strong>pi_sum</strong></td>
<td>0.83</td>
<td>1.00</td>
<td>21.07</td>
<td>14.45</td>
<td>1.28</td>
<td>260.28</td>
<td>1.27</td>
<td>2.15</td>
<td>1.23</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>rand_mat_stat</strong></td>
<td>0.99</td>
<td>1.74</td>
<td>22.29</td>
<td>16.88</td>
<td>9.82</td>
<td>30.44</td>
<td>6.20</td>
<td>2.81</td>
<td>8.23</td>
<td>3.71</td>
</tr>
<tr>
<td><strong>rand_mat_mul</strong></td>
<td>4.05</td>
<td>1.09</td>
<td>1.08</td>
<td>1.63</td>
<td>1.12</td>
<td>1.06</td>
<td>1.13</td>
<td>14.58</td>
<td>8.45</td>
<td>1.23</td>
</tr>
</tbody>
</table>
Compilation technology
Julia Compilation & Introspection

1. Parse source into syntax tree
2. Expand macros
3. Lower syntax tree
4. Type Inference
5. Build LLVM code
6. Optimize LLVM code
7. Emit machine code
Julia Compilation & Introspection

Parse source into syntax tree

Expand macros

Lower syntax tree

Type Inference

Build LLVM code

Optimize LLVM code

Emit machine code

code_lowered
code_typed
code_llvm
code_llvm
code_native
function bar(x)
    y = 1
    x-y
end

julia> code_lowered(bar,(Int,))
function bar(x)
    y = 1
    x-y
end

julia> code_lowered(bar,(Int,))
1-element Array{Any,1}:
  :($Expr(:lambda, {:x}, {{:y},{:x,:Any,0},{:y,:Any,18}},{}),
  :(begin  # /tmp/bar.jl, line 2:
   y = 1 # line 3:
   return x - y
end)))
function bar(x)
    y = 1
    x-y
end

julia> code_typed(bar,(Int,))
function bar(x)
    y = 1
    x - y
end

julia> code_typed(bar, (Int,))
1-element Array{Any,1}:
    :($(Expr(:lambda, {:x}, {{:y},{{:x,Int64,0},{:y,Int64,18}},{}},
    :(begin  # /tmp/bar.jl, line 2:
        y = 1 # line 3:
        return
        (top(box))(Int64,(top(sub_int))(x::Int64,y::Int64)::Int64
        end::Int64)))))
function bar(x)
    y = 1
    x-y
end

julia> code_llvm(bar,(Int,))
function bar(x)
    y = 1
    x-y
end

julia> code_llvm(bar,(Int,))
define i64 @julia_bar_20432(i64) {
top:
    %1 = add i64 %0, -1, !dbg !998
    ret i64 %1, !dbg !998
}
function bar(x)
    y = 1
    x-y
end

julia> code_native (bar,(Int,))
code_native

function bar(x)
    y = 1
    x-y
end

julia> code_native (bar,(Int,))
 .text
Filename: /tmp/bar.jl
Source line: 3
    push    RBP
    mov     RBP, RSP
Source line: 3
    lea     RAX, QWORD PTR [RDI - 1]
    pop     RBP
    ret
How to Get Performance?

Try to generate code like a static compiler does!

- Machine-friendly concrete types
- Avoid run-time dispatch
- Inline function calls
- Avoid boxing
C Semantics for Variables

```c
int foo() {
    int x;
    x = 2;
    x = 3.1;
    return x;
}
```

A variable is a **location** in memory.
C Semantics for Variables

```c
int foo() {
    int x;
    x = 2;
    x = 3.1;
    return x;
}
```

A variable is a **location** in memory.
C Semantics for Variables

```c
int foo() {
    int x;
    x = 2;
    x = 3.1;
    return x;
}
```

A variable is a **location** in memory.
C Semantics for Variables

```c
int foo() {
    int x;
    x = 2;
    x = 3.1;
    return x;
}
```

A variable is a **location** in memory.
C Semantics for Variables

```c
int foo() {
    int x;
    x = 2;
    x = 3.1;
    return x;
}
```

A variable is a **location** in memory.

X

```
3
```
A variable is a **location** in memory.
C vs. Julia Semantics for Variables

int foo() {
    int x;
    x = 2;
    x = 3.1;
    return x;
}

function foo() {
    local x
    x = 2
    x = 3.1
    x
}

A variable is a location in memory.

A variable is a name bound to a value.
C vs. Julia Semantics for Variables

A variable is a **location** in memory.

A variable is a **name** bound to a value.
C vs. Julia Semantics for Variables

A variable is a **location** in memory.

A variable is a **name** bound to a value.
C vs. Julia Semantics for Variables

```c
int foo() {
    int x;
    x = 2;
    x = 3.1;
    return x;
}
```

```julia
function foo()
    local x
    x = 2
    x = 3.1
    x
end
```

A variable is a **location** in memory.

A variable is a **name** bound to a value.
Concrete vs. Non-Concrete Types

Concrete

- Int
- Vector{Int}
- Tuple{Int, Float32}

```haskell
type Foo
  x::Int
  y::Float32
end
```
Concrete vs. Non-Concrete Types

Non-Concrete (require boxing)

Any

Integer

Union{Int32,Int64}

Vector{T}

Concrete

type Foo

Vector{Int}

Int

type Foo

Tuple{Int,Float32}

x::Int

y::Float32

end
Boxing

Used when compiler cannot predict that object has a specific concrete immutable type.

```plaintext
function bar()
    x = 2
    x = 3.1
    y = 4.0
    x+y
end
```

Diagram:

- **Boxed value**
- **Unboxed value**
Boxing

Used when compiler cannot predict that object has a specific concrete immutable type.

```haskell
function bar()
    x = 2
    x = 3.1
    y = 4.0
    x+y
end
```
Boxing

Used when compiler cannot predict that object has a specific concrete immutable type.

```
function bar()
  x = 2
  x = 3.1
  y = 4.0
  x+y
end
```
Boxing

Used when compiler cannot predict that object has a specific concrete immutable type.

```plaintext
function bar()
    x = 2
    x = 3.1
    y = 4.0
    x+y
end
```

Unboxed value

Boxed value

Indirection

Heap allocation

Run-time dispatch of +
Run-time Dispatch is Slow

Must find function that is the best match
- Have to look at types of all the arguments
- Run-time method caches help some

Further injury: Prevents inlining
How C++ Solves the Problem

Programmer declares types of variables
- C++11: auto for trivially inferable cases

Restrict multiple dispatch to compilation time

Restrict run-time dispatch to first argument (this)

Programmer declares which functions use single dispatch
- Enables compile-time computation of virtual function tables
- Run-time tags restricted to types with virtual functions
Julia Solution

Type inference

Type-stable primitives

Specialization of functions

Concrete types cannot be subtyped

Care from the programmer to enable inference of concrete types
Type-Stability Problem in Python 2.7

```python
def foo(n):
    i = 10
    for j in range(n):
        i = i*i*i
        print type(i), i

foo(3)  # What does this call print?
```
Type-Stability Problem in Python 2.7

def foo(n):
    i = 10
    for j in range(n):
        i = i*i*i
        print type(i), i

foo(3)  # What does this call print?

<type 'int'> 1000
<type 'int'> 1000000000
<type 'long'> 1000000000000000000000000000

Correct answers, but at a performance cost imposed on everyone.
Julia Primitives Are Type Stable

```julia
function foo(n)
    i = 10
    for j=1:n
        i = i*i*i
        println(typeof(i)," ",i)
    end
end
foo(3)
```
Julia Primitives Are Type Stable

function foo(n)
    i = 10
    for j=1:n
        i = i*i*i
        println(typeof(i)," ",i)
    end
end

foo(3)

Int64 1000
Int64 1000000000
Int64 -6930898827444486144

Int64 * Int64 always returns an Int64. Answer correct modulo $2^{64}$
Code Specialization

JIT generates specialized code for functions

- Code specialized for actual (concrete) types of arguments
  - Like making every function a template function in C++.
  - Specialization explosion seems to not be a problem in scientific codes.
- Prohibition against “subtyping of concrete types” helps.
Example

```plaintext
function axpy(a, x, y)
    for i=1:length(x)
        y[i] += a*x[i]
    end
end

n = 1000
u = rand(Float32,n)
v = rand(Float32,n)
w = rand(Float64,n)
axpy(2, u, v) # Generates specialization #1
axpy(3, u, v) # Reuses specialization #1
axpy(2, u, w) # Generates specialization #2
```
Example

function axpy( a, x, y )
    for i=1:length(x)
        y[i] += a*x[i]
    end
end

n = 1000
u = rand(Float32,n)
v = rand(Float32,n)
w = rand(Float64,n)
axpy(2, u, v)     // Generates specialization #1
axpy(3, u, v)     // Reuses specialization #1
axpy(2, u, w)     // Generates specialization #2
resize!(u,500)
resize!(v,500)
axpy(4, u, v)     // Reuses specialization #1
Composite Types Not Specialized

type Circle
  x
  y
  r
end

Circle is a concrete type. But x, y, and r known to be Any.
Composite Types Not Specialized

type Circle
  x
  y
  r
end

Circle is a concrete type. But x, y, and r known to be Any.

Requires boxing.
Still Not Concrete

type Circle
  x :: Real
  y :: Real
  r :: Real
end

Still requires boxing, since Real is abstract type.

Boxes known to hold some concrete subtype of Real.
Concrete

```plaintext
type Circle
  x :: Float64
  y :: Float64
  r :: Float64
end
```

<table>
<thead>
<tr>
<th></th>
<th>64-bit float</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td></td>
</tr>
</tbody>
</table>
Generalize with Parametric Types

define Circle{T<:Real}
    x :: T
    y :: T
    r :: T
end
Generalize with Parametric Types

```plaintext
type Circle{T<:Real}
    x :: T
    y :: T
    r :: T
end
```

Circle{Float64}

<table>
<thead>
<tr>
<th></th>
<th>64-bit float</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>64-bit float</td>
</tr>
<tr>
<td>r</td>
<td>64-bit float</td>
</tr>
</tbody>
</table>
Generalize with Parametric Types

type Circle{T<:Real}
    x :: T
    y :: T
    r :: T
end

Circle{Float64}

<table>
<thead>
<tr>
<th>x</th>
<th>64-bit float</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>64-bit float</td>
</tr>
<tr>
<td>r</td>
<td>64-bit float</td>
</tr>
</tbody>
</table>

Circle{Real}

<table>
<thead>
<tr>
<th>x</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>data</td>
</tr>
<tr>
<td>r</td>
<td>data</td>
</tr>
</tbody>
</table>
## Compiler’s Knowledge of Types

<table>
<thead>
<tr>
<th>Context</th>
<th>Compiler Treatment</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter</td>
<td>usually known exactly</td>
</tr>
<tr>
<td>global const variable</td>
<td>usually known exactly</td>
</tr>
<tr>
<td>local variable</td>
<td>inferred</td>
</tr>
<tr>
<td>return value</td>
<td>inferred</td>
</tr>
<tr>
<td>fields of structures</td>
<td>as declared</td>
</tr>
<tr>
<td>global variable</td>
<td>unknown</td>
</tr>
</tbody>
</table>
Be Nice to Type Inference!

The big performance issue in Julia.

- Julia functions are polymorphic
- Hardware is monomorphic.

Impact of type uncertainty:
Be Nice to Type Inference!

The big performance issue in Julia.

- Julia functions are polymorphic
- Hardware is monomorphic.

Impact of type uncertainty:

- Boxing
  - Heap allocation
  - Garbage collection (GC)
Be Nice to Type Inference!

The big performance issue in Julia.

- Julia functions are polymorphic
- Hardware is monomorphic.

Impact of type uncertainty:

- Boxing
  - Heap allocation
  - Garbage collection (GC)
- Run-time dispatch of calls
  - Table scanning
  - Call cannot be inlined.
Type Inference Algorithm

Subtyping relationships form a lattice

- Any at top
- Concrete types just above bottom

Figure adapted from figure 3-1 of Jeff Bezanson’s thesis.
Type Inference Algorithm

Subtyping relationships form a lattice

- Any at top
- Concrete types just above bottom

**Forward** flow analysis

- Join over all paths
- Not flow-sensitive within a function

Interprocedural

- Context sensitive for call sites

Figure adapted from figure 3-1 of Jeff Bezanson's thesis.
Inference Example

function qux(x)
  if x ≥ 0
    y = x
  else
    y = 0
  end
  1 + y
end

julia> code_llvm(qux, (Int,))
Inference Example

```julia
function qux(x)
  if x >= 0
    y = x
  else
    y = 0
  end
  1 + y
end
```

```julia
code_llvm(qux,(Int,))
```

```
define i64 @julia_qux_21087(i64) {
  top:
    %1 = icmp slt i64 %0, 0
    br i1 %1, label %L1, label %if
  if:
    ; preds = %top
    %phitmp = add i64 %0, 1
    br label %L1
  L1:
    ; preds = %if, %top
    %y.0 = phi i64 [ %phitmp, %if ], [ 1, %top ]
    ret i64 %y.0
}
```
Similar?

```julia
function qux(x)
    if x ≥ 0
        y = x
    else
        y = 0
    end
    1 + y
end
```

```julia
julia> code_llvm(qux, (Float32,))
```
function qux(x)
    if x≥0
        y=x
    else
        y=0
    end
end

julia> code_llvm(qux,(Float32,))

define %jl_value_t* @julia_qux_21069(float) {
top:
    %1 = alloca [5 x %jl_value_t*], align 8
    %2 = getelementptr inbounds [5 x %jl_value_t*] %1, i64 0, i64 0
    %3 = bitcast [5 x %jl_value_t*] %1 to i64*
    store i64 6, i64* %3, align 8
    %4 = getelementptr [5 x %jl_value_t*] %1, i64 0, i64 1
    %5 = bitcast %jl_value_t** %4 to %jl_value_t***
    %6 = load %jl_value_t*** @jl_pgcstack, align 8
    store %jl_value_t*** %6, %jl_value_t*** @jl_pgcstack, align 8
    store %jl_value_t*** %5, %jl_value_t*** @jl_pgcstack, align 8
    %7 = getelementptr [5 x %jl_value_t*] %1, i64 0, i64 3
    store %jl_value_t* null, %jl_value_t** %7, align 8
    %8 = getelementptr [5 x %jl_value_t*] %1, i64 0, i64 4
    store %jl_value_t* null, %jl_value_t** %8, align 8
    %9 = fcmp ult float %0, 0.000000e+00
    br i1 %9, label %if, label %top
    if: ; preds = %top
        %10 = call %jl_value_t* @jl_box_float32(float %0)
        br label %L1
    L1: ; preds = %if, %top
        %11 = call %jl_value_t* @jl_apply_generic(%jl_value_t* inttoptr (i64 140243857278128 to %jl_value_t*), %jl_value_t** %7, i32 2)
        %12 = load %jl_value_t*** %5, align 8
        store %jl_value_t** %12, %jl_value_t*** @jl_pgcstack, align 8
        ret %jl_value_t* %11
}
function qux(x)
    if x≥0
        y = x
    else
        y = 0
    end
end

julia> code_llvm(qux,(Float32,))

define %jl_value_t* @julia_qux_21069(float) {
    top:
        %1 = alloca [5 x %jl_value_t*], align 8
        %.sub = getelementptr inbounds [5 x %jl_value_t*] %1, i64 0, i64 0
        %2 = getelementptr [5 x %jl_value_t*] %1, i64 0, i64 2
        %3 = bitcast [5 x %jl_value_t*] %1 to i64*
        store i64 6, i64* %3, align 8
        %4 = getelementptr [5 x %jl_value_t*] %1, i64 0, i64 1
        %5 = bitcast %jl_value_t** %4 to %jl_value_t***
        %6 = load %jl_value_t*** @jl_pgcstack, align 8
        store %jl_value_t** %6, %jl_value_t*** @jl_pgcstack, align 8
        store %jl_value_t** %.sub, %jl_value_t*** @jl_pgcstack, align 8
        %7 = getelementptr [5 x %jl_value_t*] %1, i64 0, i64 3
        store %jl_value_t* null, %jl_value_t** %7, align 8
        %8 = getelementptr [5 x %jl_value_t*] %1, i64 0, i64 4
        store %jl_value_t* null, %jl_value_t** %8, align 8
        store %jl_value_t* null, %jl_value_t** %2, align 8
        %9 = fcmp ult float %0, 0.000000e+00
        br i1 %9, label %L1, label %if
    if:                                           ; preds = %top
        %10 = call %jl_value_t* @jl_box_float32(float %0)
        br label %L1
    L1:                                           ; preds = %if, %top
        %storemerge = phi %jl_value_t* [%10, %if], [inttoptr (i64 140243853189200 to %jl_value_t*), %top]
        store %jl_value_t* %storemerge, %jl_value_t** %2, align 8
        store %jl_value_t* %storemerge, %jl_value_t** %7, align 8
        store %jl_value_t* inttoptr (i64 140243853189248 to %jl_value_t*), %jl_value_t** %8, align 8
        %11 = call %jl_value_t* @jl_apply_generic(%jl_value_t* inttoptr (i64 140243857278 to %jl_value_t*), %jl_value_t** %7, i32 2)
        %12 = load %jl_value_t*** %5, align 8
        store %jl_value_t** %12, %jl_value_t*** @jl_pgcstack, align 8
        ret %jl_value_t* %11
    GC-related stuff%
function qux(x)
    if x ≥ 0
        y = x
    else
        y = 0
    end
    1 + y
end

julia> code_llvm(qux,(Float32,))

define %jl_value_t* @julia_qux_21069(float) {
    top:
        %1 = alloca [5 x %jl_value_t*], align 8
        %sub = getelementptr inbounds [5 x %jl_value_t*] %1, i64 0, i64 0
        %2 = getelementptr [5 x %jl_value_t*] %1, i64 0, i64 1
        %3 = bitcast [5 x %jl_value_t*] %1 to i64*
        store i64 6, i64* %3, align 8
        %4 = getelementptr [5 x %jl_value_t*] %1, i64 0, i64 2
        %5 = bitcast %jl_value_t** %4 to %jl_value_t***
        %6 = load %jl_value_t*** @jl_pgcstack, align 8
        store %jl_value_t** %6, %jl_value_t*** @jl_pgcstack, align 8
        store %jl_value_t* inttoptr (i64 140243853189248 to %jl_value_t*), %jl_value_t** %2, align 8
        %7 = fcmp ult float %0, 0.000000e+00
        br i1 %7, label %if
    if:                                        ; preds = %top
        %10 = call %jl_value_t* @jl_box_float32(float %0)
        br label %L1
    L1:                                       ; preds = %if, %top
        %storemerge = phi %jl_value_t* [%10, %if ], [ inttoptr (i64 140243853189200 to %jl_value_t*), %top]
        store %jl_value_t* %storemerge, %jl_value_t** %2, align 8
        store %jl_value_t* %storemerge, %jl_value_t** %7, align 8
        store %jl_value_t* inttoptr (i64 140243853189248 to %jl_value_t*), %jl_value_t** %8, align 8
        %11 = call %jl_value_t* @jl_apply_generic(%jl_value_t* inttoptr (i64 1402438572784 to %jl_value_t*), %jl_value_t** %7, i32 2)
        %12 = load %jl_value_t*** %5, align 8
        store %jl_value_t** %12, %jl_value_t*** @jl_pgcstack, align 8
        ret %jl_value_t* %11
    ret
Define a function `qux(x)`:

```plaintext
function qux(x)
    if x≥0
        y=x
    else
        y=0
    end
    1+y
end
```

Using `code_llvm` to convert this Julia code to LLVM IR:

```plaintext
define %jl_value_t* @julia_qux_21069(float) {
  top:
    %1 = alloca [5 x %jl_value_t*], align 8
    %2 = getelementptr inbounds [5 x %jl_value_t*]* %1, i64 0, i64 0
    %3 = bitcast [5 x %jl_value_t*]* %1 to i64*
    store i64 6, i64* %3, align 8
    %4 = getelementptr [5 x %jl_value_t*]* %1, i64 0, i64 1
    %5 = bitcast %jl_value_t** %4 to %jl_value_t***
    %6 = load %jl_value_t*** @jl_pgcstack, align 8
    store %jl_value_t** %6, %jl_value_t*** @jl_pgcstack, align 8
    store %jl_value_t** %2, %jl_value_t*** @jl_pgcstack, align 8
    %7 = getelementptr [5 x %jl_value_t*]* %1, i64 0, i64 3
    store %jl_value_t* null, %jl_value_t** %7, align 8
    %8 = getelementptr [5 x %jl_value_t*]* %1, i64 0, i64 4
    store %jl_value_t* null, %jl_value_t** %8, align 8
    store %jl_value_t* null, %jl_value_t** %2, align 8
    %9 = fcmp ult float %0, 0.000000e+00
    br i1 %9, label %L1, label %if

  if:                                               ; preds = %top
    %10 = call %jl_value_t* @jl_box_float32(float %0)
    br label %L1

  L1:                                               ; preds = %if, %top
    %storemerge = phi %jl_value_t* [ %10, %if ], [ inttoptr (i64 140243853189200 to %jl_value_t), %top]
    store %jl_value_t* %storemerge, %jl_value_t** %2, align 8
    store %jl_value_t* %storemerge, %jl_value_t** %7, align 8
    store %jl_value_t* inttoptr (i64 140243853189248 to %jl_value_t*), %jl_value_t** %8, align 8
    %11 = call %jl_value_t* @jl_apply_generic(%jl_value_t* inttoptr (i64 140243853189248 to %jl_value_t*), %jl_value_t** %7, i32 2)
    %12 = load %jl_value_t*** %5, align 8
    store %jl_value_t** %12, %jl_value_t*** @jl_pgcstack, align 8
    ret %jl_value_t* %11
}
```

### GC-related stuff
- Boxing `x`
- Run-time dispatch
Root Problem

```julia
function qux(x)
    if x≥0
        y=x
    else
        y=0
    end
    1+y
end
```

```julia
code_warntype(qux,(Float32,))
    ....
    y = x::Float32
    goto 1
    0:  # line 5:
    y = 0
    1:  # line 7:
        return 1 +
    y::Union{Int64,Float32}::Union{Int64,Float32}
end::Union{Int64,Float32}
```

code_warntype is Julia 0.4 feature
Possible Fixes

```julia
function qux(x)
    if x ≥ 0
        y = x
    else
        y = zero(x)
    end
    1 + y
end

function qux{T}(x::T)
    if x ≥ 0
        y = x
    else
        y = zero(T)
    end
    1 + y
end

function qux{T}(x::T)
    if x ≥ 0
        y = x
    else
        y = convert(T,0)
    end
    1 + y
end

function qux{T}(x::T)
    if x ≥ 0
        y = x
    else
        y = T(0)
    end
    1 + y
end
```

Julia 0.4
Issue is NOT Mixed-Mode Arithmetic

Performance problems relate to predictability

Mix-mode arithmetic works fine

- Julia has natural type-promotion system for numerical operations
- Implemented by the standard library (not the compiler)
- Extensible to user-defined operations
Global Variables

No type inference for reassignable global variables.

- Julia is dynamic -- more assignments might be added later.

Work arrounds

- Avoid global variables
- Declare single-assignment global variables const
- Use explicit type-check or force conversion
- Pass as parameter to helper function
Vectorization

Program transformation for exploiting SIMD units

- Not to be confused with other use of “vectorization” to mean array-oriented operations.

```
10 10 20 20 30 30 40 40
+  
1 2 3 4 5 6 7 8
```

SIMD Arithmetic

```plaintext
11 12 23 24 35 36 47 48
```
Vectorization of a Loop

function axpy( a, x, y )
    @simd for i=1:length(x)
        @inbounds y[i] = y[i]+a*x[i]
    end
end
Vectorization of a Loop

```plaintext
function axpy( a, x, y )
  @simd for i=1:length(x)
    @inbounds y[i] = y[i]+a*x[i]
  end
end

function axpy( a::Float32, x::Array{Float32,1}, y::Array{Float32,1} )
  @inbounds for i=1:4:length(x)-3
    # Four logical iterations per physical iteration
    t1 = (x[i],x[i+1],x[i+2],x[i+3])  # Load tuple
    t2 = (y[i],y[i+1],y[i+2],y[i+3])  # Load tuple
    t3 = a*t1                         # Scalar times tuple
    t4 = t2+t3                        # Tuple add
    (y[i],y[i+1],y[i+2],y[i+3]) = t4  # Tuple store
  end
  ... Scalar loop for remaining iterations ...
end

Note: example assumes tuple math exists.
```
Serial Order of Evaluation

\[ t_{11} = x[1] \]
\[ t_{12} = x[2] \]
\[ t_{13} = x[3] \]
\[ t_{14} = x[4] \]
\[ t_{21} = y[1] \]
\[ t_{22} = y[2] \]
\[ t_{23} = y[3] \]
\[ t_{24} = y[4] \]
\[ t_{31} = a \cdot t_{11} \]
\[ t_{32} = a \cdot t_{12} \]
\[ t_{33} = a \cdot t_{13} \]
\[ t_{34} = a \cdot t_{14} \]
\[ t_{41} = t_{21} + t_{31} \]
\[ t_{42} = t_{22} + t_{32} \]
\[ t_{43} = t_{23} + t_{33} \]
\[ t_{44} = t_{24} + t_{34} \]
\[ y[1] = t_{41} \]
\[ y[2] = t_{42} \]
\[ y[3] = t_{43} \]
\[ y[4] = t_{44} \]
Serial Order of Evaluation

\[ \begin{align*}
t_1_1 &= x[1] \\
t_1_2 &= x[2] \\
t_1_3 &= x[3] \\
t_1_4 &= x[4] \\
t_2_1 &= y[1] \\
t_2_2 &= y[2] \\
t_2_3 &= y[3] \\
t_2_4 &= y[4] \\
t_3_1 &= a \times t_1_1 \\
t_3_2 &= a \times t_1_2 \\
t_3_3 &= a \times t_1_3 \\
t_3_4 &= a \times t_1_4 \\
t_4_1 &= t_2_1 + t_3_1 \\
t_4_2 &= t_2_2 + t_3_2 \\
t_4_3 &= t_2_3 + t_3_3 \\
t_4_4 &= t_2_4 + t_3_4 \\
y[1] &= t_4_1 \\
y[2] &= t_4_2 \\
y[3] &= t_4_3 \\
y[4] &= t_4_4
\end{align*} \]
Serial Order of Evaluation

\[
\begin{align*}
t_1_1 &= x[1] \\
t_2_1 &= y[1] \\
t_3_1 &= a * t_1_1 \\
t_4_1 &= t_2_1 + t_3_1 \\
y[1] &= t_4_1
\end{align*}
\]

\[
\begin{align*}
t_1_2 &= x[2] \\
t_2_2 &= y[2] \\
t_3_2 &= a * t_1_2 \\
t_4_2 &= t_2_2 + t_3_2 \\
y[2] &= t_4_2
\end{align*}
\]

\[
\begin{align*}
t_1_3 &= x[3] \\
t_2_3 &= y[3] \\
t_3_3 &= a * t_1_3 \\
t_4_3 &= t_2_3 + t_3_3 \\
y[3] &= t_4_3
\end{align*}
\]

\[
\begin{align*}
t_1_4 &= x[4] \\
t_2_4 &= y[4] \\
t_3_4 &= a * t_1_4 \\
t_4_4 &= t_2_4 + t_3_4 \\
y[4] &= t_4_4
\end{align*}
\]
Serial Order of Evaluation

\[ \begin{align*}
  t_1^1 &= x[1] \\
  t_1^2 &= x[2] \\
  t_1^3 &= x[3] \\
  t_1^4 &= x[4] \\
  t_2^1 &= y[1] \\
  t_2^2 &= y[2] \\
  t_2^3 &= y[3] \\
  t_2^4 &= y[4] \\
  t_3^1 &= a \times t_{11} \\
  t_3^2 &= a \times t_{12} \\
  t_3^3 &= a \times t_{13} \\
  t_3^4 &= a \times t_{14} \\
  t_4^1 &= t_{21} + t_{31} \\
  t_4^2 &= t_{22} + t_{32} \\
  t_4^3 &= t_{23} + t_{33} \\
  t_4^4 &= t_{24} + t_{34} \\
  y[1] &= t_4^1 \\
  y[2] &= t_4^2 \\
  y[3] &= t_4^3 \\
  y[4] &= t_4^4
\end{align*} \]
Serial Order of Evaluation

\[ t_1 = x[1] \]
\[ t_2 = y[1] \]
\[ t_3 = a \times t_1 \]
\[ t_4 = t_2 + t_3 \]
\[ y[1] = t_4 \]

\[ t_1 = x[2] \]
\[ t_2 = y[2] \]
\[ t_3 = a \times t_2 \]
\[ t_4 = t_2 + t_3 \]
\[ y[2] = t_4 \]

\[ t_1 = x[3] \]
\[ t_2 = y[3] \]
\[ t_3 = a \times t_3 \]
\[ t_4 = t_2 + t_3 \]
\[ y[3] = t_4 \]

\[ t_1 = x[4] \]
\[ t_2 = y[4] \]
\[ t_3 = a \times t_4 \]
\[ t_4 = t_2 + t_3 \]
\[ y[4] = t_4 \]
Serial Order of Evaluation

\[ t_{1,1} = x[1] \quad t_{1,2} = x[2] \quad t_{1,3} = x[3] \quad t_{1,4} = x[4] \]

\[ t_{2,1} = y[1] \quad t_{2,2} = y[2] \quad t_{2,3} = y[3] \quad t_{2,4} = y[4] \]

\[ t_{3,1} = a \cdot t_{1,1} \quad t_{3,2} = a \cdot t_{1,2} \quad t_{3,3} = a \cdot t_{1,3} \quad t_{3,4} = a \cdot t_{1,4} \]

\[ t_{4,1} = t_{2,1} + t_{3,1} \quad t_{4,2} = t_{2,2} + t_{3,2} \quad t_{4,3} = t_{2,3} + t_{3,3} \quad t_{4,4} = t_{2,4} + t_{3,4} \]

\[ y[1] = t_{4,1} \quad y[2] = t_{4,2} \quad y[3] = t_{4,3} \quad y[4] = t_{4,4} \]
Serial Order of Evaluation

\[ t_{1_1} = x[1] \]
\[ t_{1_2} = x[2] \]
\[ t_{1_3} = x[3] \]
\[ t_{1_4} = x[4] \]

\[ t_{2_1} = y[1] \]
\[ t_{2_2} = y[2] \]
\[ t_{2_3} = y[3] \]
\[ t_{2_4} = y[4] \]

\[ t_{3_1} = a \times t_{1_1} \]
\[ t_{3_2} = a \times t_{1_2} \]
\[ t_{3_3} = a \times t_{1_3} \]
\[ t_{3_4} = a \times t_{1_4} \]

\[ t_{4_1} = t_{2_1} + t_{3_1} \]
\[ t_{4_2} = t_{2_2} + t_{3_2} \]
\[ t_{4_3} = t_{2_3} + t_{3_3} \]
\[ t_{4_4} = t_{2_4} + t_{3_4} \]

\[ y[1] = t_{4_1} \]
\[ y[2] = t_{4_2} \]
\[ y[3] = t_{4_3} \]
\[ y[4] = t_{4_4} \]
Serial Order of Evaluation

\[ t_1 = x[1] \]
\[ t_2 = x[2] \]
\[ t_3 = x[3] \]
\[ t_4 = x[4] \]

\[ t_1 = y[1] \]
\[ t_2 = y[2] \]
\[ t_3 = y[3] \]
\[ t_4 = y[4] \]

\[ t_1 = a \times t_1 \]
\[ t_2 = a \times t_2 \]
\[ t_3 = a \times t_3 \]
\[ t_4 = a \times t_4 \]

\[ t_1 = t_2 + t_3 \]
\[ t_2 = t_2 + t_3 \]
\[ t_3 = t_2 + t_3 \]
\[ t_4 = t_2 + t_3 \]

\[ y[1] = t_4 \]
\[ y[2] = t_4 \]
\[ y[3] = t_4 \]
\[ y[4] = t_4 \]
Serial Order of Evaluation

\[
\begin{align*}
t_1_1 &= x[1] \\
t_2_1 &= y[1] \\
t_3_1 &= a * t_1_1 \\
t_4_1 &= t_2_1 + t_3_1 \\
y[1] &= t_4_1 \\

t_1_2 &= x[2] \\
t_2_2 &= y[2] \\
t_3_2 &= a * t_1_2 \\
t_4_2 &= t_2_2 + t_3_2 \\
y[2] &= t_4_2 \\

t_1_3 &= x[3] \\
t_2_3 &= y[3] \\
t_3_3 &= a * t_1_3 \\
t_4_3 &= t_2_3 + t_3_3 \\
y[3] &= t_4_3 \\

t_1_4 &= x[4] \\
t_2_4 &= y[4] \\
t_3_4 &= a * t_1_4 \\
t_4_4 &= t_2_4 + t_3_4 \\
y[4] &= t_4_4
\end{align*}
\]
Serial Order of Evaluation

\[ t_{11} = x[1] \]
\[ t_{21} = y[1] \]
\[ t_{31} = a \cdot t_{11} \]
\[ t_{41} = t_{21} + t_{31} \]
\[ y[1] = t_{41} \]

\[ t_{12} = x[2] \]
\[ t_{22} = y[2] \]
\[ t_{32} = a \cdot t_{12} \]
\[ t_{42} = t_{22} + t_{32} \]
\[ y[2] = t_{42} \]

\[ t_{13} = x[3] \]
\[ t_{23} = y[3] \]
\[ t_{33} = a \cdot t_{13} \]
\[ t_{43} = t_{23} + t_{33} \]
\[ y[3] = t_{43} \]

\[ t_{14} = x[4] \]
\[ t_{24} = y[4] \]
\[ t_{34} = a \cdot t_{14} \]
\[ t_{44} = t_{24} + t_{34} \]
\[ y[4] = t_{44} \]
Serial Order of Evaluation

\[
\begin{align*}
t_1 &= x[1] \\
t_2 &= x[2] \\
t_3 &= x[3] \\
t_4 &= x[4] \\

t_1 &= y[1] \\
t_2 &= y[2] \\
t_3 &= y[3] \\
t_4 &= y[4] \\

t_1 &= a \cdot t_1 \\
t_2 &= a \cdot t_2 \\
t_3 &= a \cdot t_3 \\
t_4 &= a \cdot t_4 \\

t_1 &= t_1 + t_3 \\
t_2 &= t_2 + t_3 \\
t_3 &= t_2 + t_3 \\
t_4 &= t_2 + t_3 \\
	y[1] &= t_1 + t_3 \\
y[2] &= t_2 + t_3 \\
y[3] &= t_3 \\
y[4] &= t_4 \\
\end{align*}
\]
Current `@simd` Order in Julia

\[
\begin{align*}
t_1_1 &= x[1] \\
t_2_1 &= y[1] \\
t_3_1 &= a \times t_1_1 \\
t_4_1 &= t_2_1 + t_3_1 \\
y[1] &= t_4_1 \\
t_1_2 &= x[2] \\
t_2_2 &= y[2] \\
t_3_2 &= a \times t_1_2 \\
t_4_2 &= t_2_2 + t_3_2 \\
y[2] &= t_4_2 \\
t_1_3 &= x[3] \\
t_2_3 &= y[3] \\
t_3_3 &= a \times t_1_3 \\
t_4_3 &= t_2_3 + t_3_3 \\
y[3] &= t_4_3 \\
t_1_4 &= x[4] \\
t_2_4 &= y[4] \\
t_3_4 &= a \times t_1_4 \\
t_4_4 &= t_2_4 + t_3_4 \\
y[4] &= t_4_4
\end{align*}
\]
Future @simd Order in Julia?

For now, do not rely on the horizontal orderings.
Implicit vs. Explicit Vectorization

Implicit vectorization

Compiler proves that transposition/reassociation is legal

OR

Inserts run-time checks
Implicit vs. Explicit Vectorization

Implicit vectorization

Compiler proves that transposition/reassociation is legal

OR

Inserts run-time checks

Explicit vectorization with @simd

Experimental feature

Programmer vouches that transposition/reassociation is okay
Programmer Responsibilities

All vectorization (currently)
- No cross-iteration dependencies
- Straight-line loop body
  - @inbounds
  - All calls inlined (be nice to type inference)
- Unit-stride subscripts

Implicit vectorization
- Just a few arrays accessed
- Use @fastmath for floating-point reductions

Explicit vectorization
- Use @simd
- Ensure there are no cross-iteration dependencies.
- Local scalars for reductions.
# Advice for Users

<table>
<thead>
<tr>
<th>Transform</th>
<th>Recommended Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant propagation</td>
<td>Compiler</td>
</tr>
<tr>
<td>Algebraic simplification</td>
<td>Compiler for integers or @fastmath&lt;br&gt;You for other floating-point</td>
</tr>
<tr>
<td>Inlining</td>
<td>Compiler usually&lt;br&gt;You can use @inline&lt;br&gt;Disable with -inline=no</td>
</tr>
<tr>
<td>Eliminating bounds checks</td>
<td>Use @inbounds&lt;br&gt;Use -check-bounds=yes to force checking</td>
</tr>
<tr>
<td>Hoisting loop invariants</td>
<td>Compiler for local scalar calculations&lt;br&gt;You for field/subscript references</td>
</tr>
<tr>
<td>Unrolling loops</td>
<td>Compiler</td>
</tr>
<tr>
<td>Vectorization</td>
<td>Compiler&lt;br&gt;You must use @inbounds&lt;br&gt;You can use @simd to assist</td>
</tr>
</tbody>
</table>
Suggested Program Structure

use types to direct control flow and protect against accidents

do loads/stores for global vars.

inferable concrete types
no global variable references
@simd loops if possible
help compiler
Key Point

Dynamic languages and performance do not have to be exclusive

- But language should be designed with performance in mind.
- Performance still requires some care from user
Performance Features not Covered

Profiling

Macros

Staged programming
Resources

Julia web site: http://julialang.org/

Julia Box: https://www.juliabox.org/

“Why We Created Julia”: http://julialang.org/blog/2012/02/why-we-created-julia/


Trademark Notice

Intel is a trademark of Intel Corporation in the U.S. and/or other countries.
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.