Programming Language Design and Performance

Jonathan Aldrich

17-396: Language Design and Prototyping

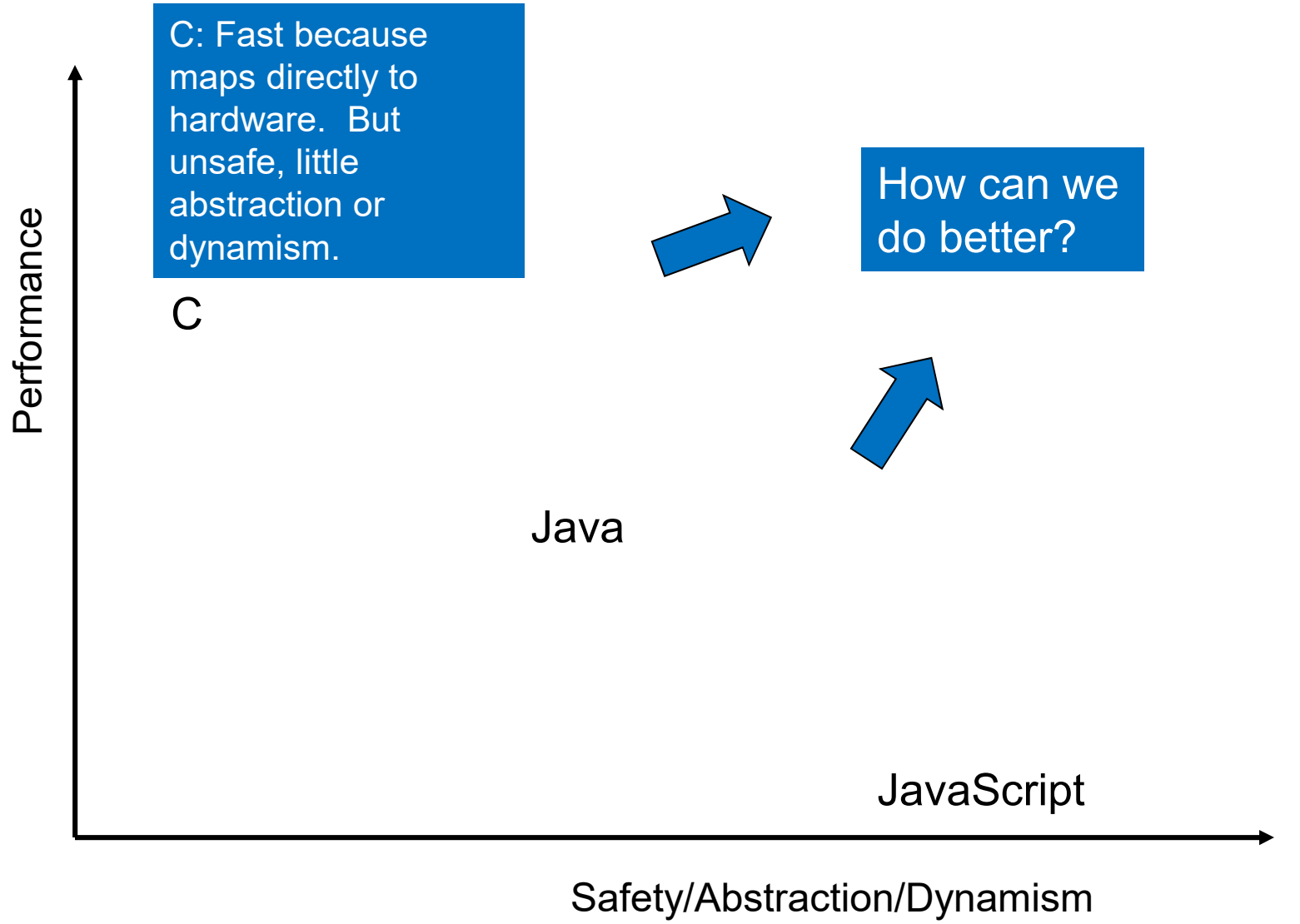
Spring 2020

institute for SOFTWARE RESEARCH

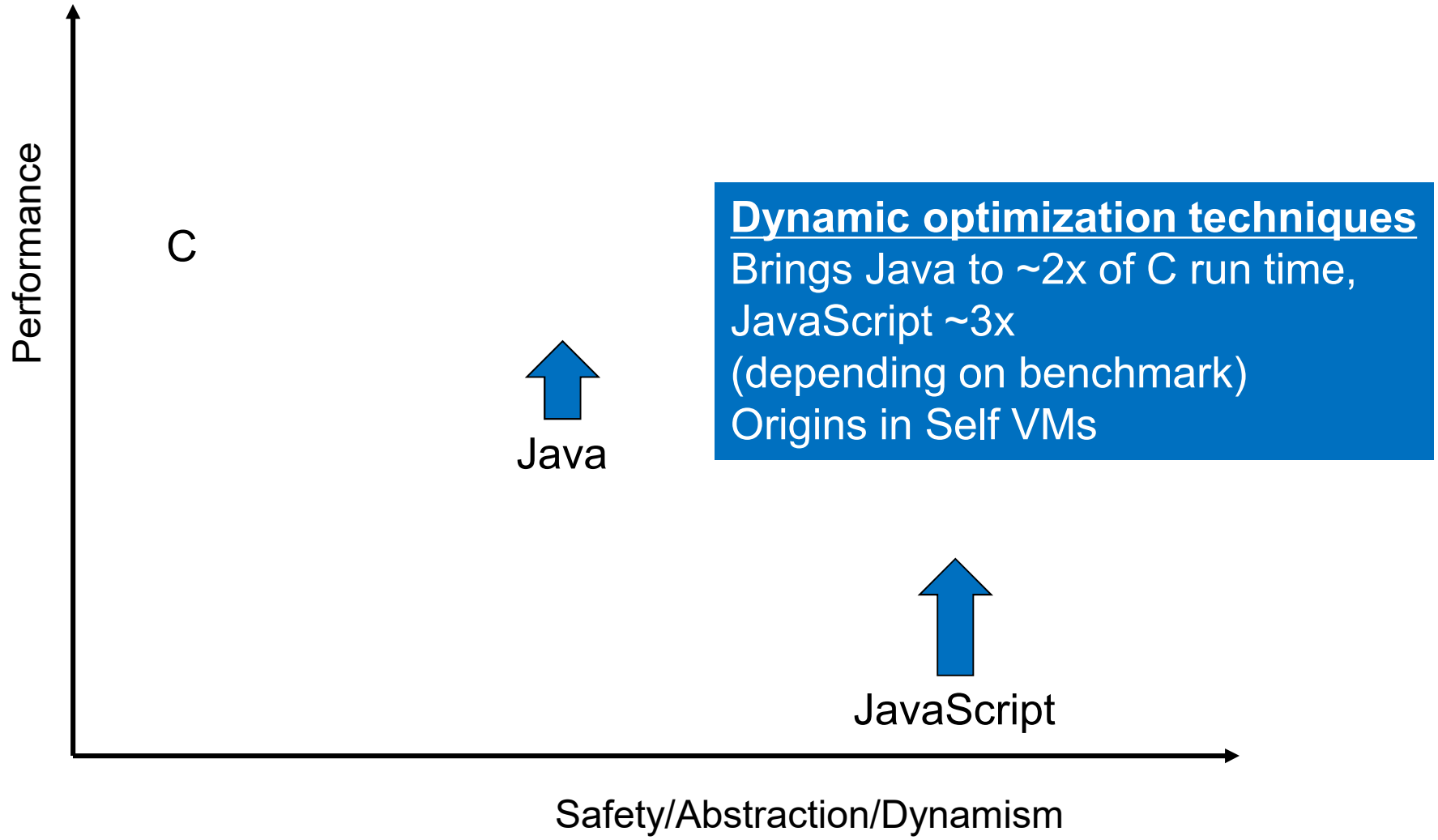
Opening discussion

- What features/tools make a language fast?

Basic Tradeoff



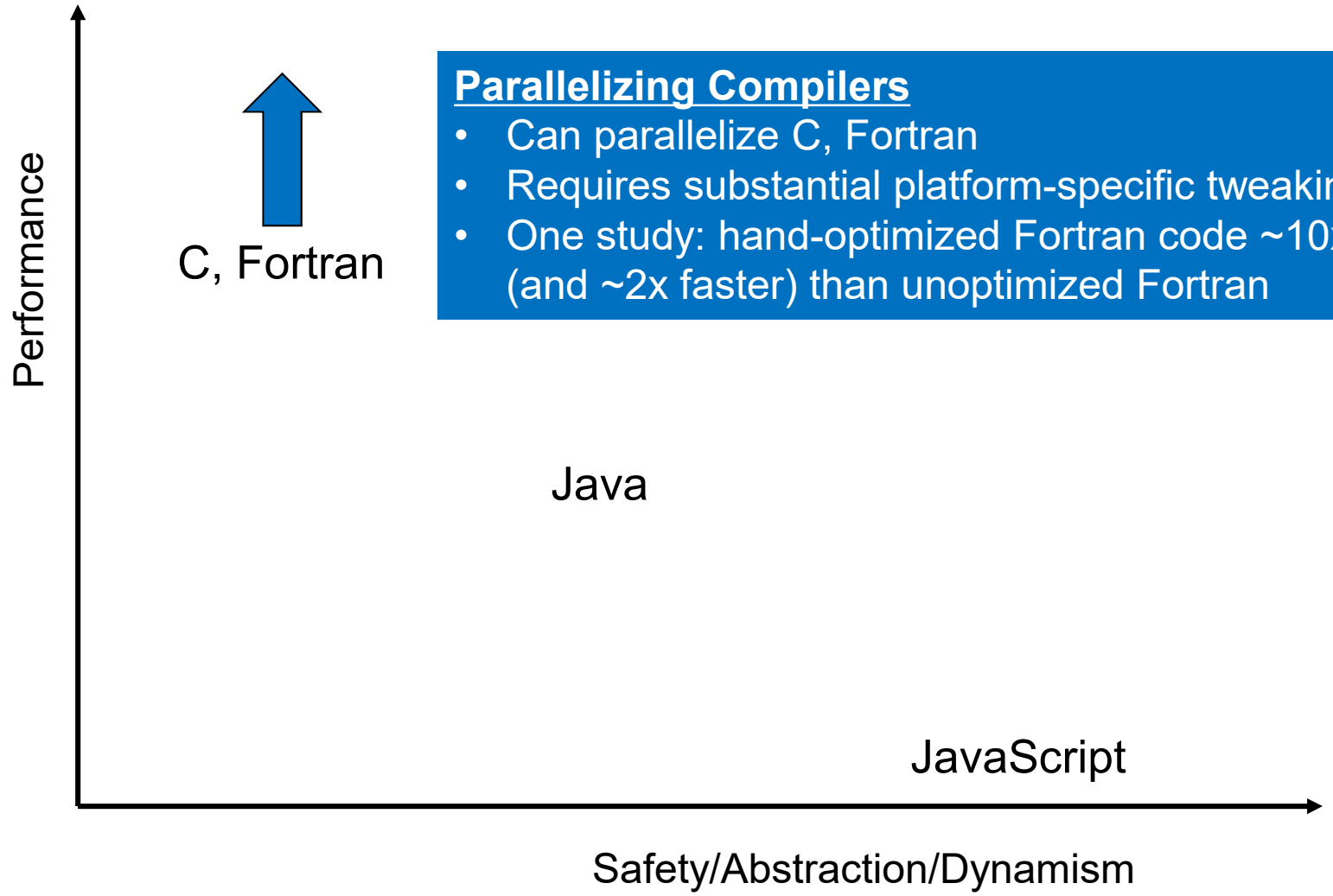
Dynamic optimization



Dynamic optimization techniques
Brings Java to ~2x of C run time, JavaScript ~3x
(depending on benchmark)
Origins in Self VMs

Source: https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Parallelizing Compilers



Parallelizing Compilers

- Can parallelize C, Fortran
- Requires substantial platform-specific tweaking
- One study: hand-optimized Fortran code ~10x larger (and ~2x faster) than unoptimized Fortran

Source: https://queue.acm.org/detail.cfm?id=1820518

Language Influence on Parallelization

- Fortran compilers assume parameters, arrays do not alias
  - Danger: this is not checked!

```
// illustrative example, in C syntax
void f(float* a, float* b, unsigned size) {
   for (unsigned i = 0; i < size; ++i)
      *a += b[i];    // Fortran can cache a in a register; C can't
}

// client code
float a[200];      // initialize to something
f(a+100, a, 200); // this would be illegal in Fortran
```

C and (especially) Fortran also benefit from mature parallelizing compilers and great libraries (BLAS, LAPACK)

Example due to euzeka at https://arstechnica.com/civis/viewtopic.php?f=20&t=631580

The Importance of Libraries

- Python: widely used for scientific computing
  - Perceived to be easy to use
  - Slow (but see PyPy, which is a lot like Truffle/Graal)
    - Dynamic, interpreted language
    - Boxed numbers (everything is a number allocated on the heap)

- Python packages for scientific computing
  - Numpy: multidimensional arrays
    - Fixed size, homogeneous, packed data (like C arrays)
    - Vectorized operations: c = a+b // *adds arrays elementwise*
  - SciPy: mathematical/scientific libraries
    - Wraps BLAS, LAPACK and others
    - Uses Numpy in interface

Julia: Performance + Usability

- Dynamic language, like Python/JavaScript
- Excellent libraries for scientific computing
  - Like Fortran, Python
- Unique performance strategy
  - Uses *multiple dispatch* to choose appropriate algorithms
    - e.g. sparse vs. full matrix multiplication; special cases for tridiagonal matrices
  - Aggressive *specialization* to overcome cost of abstraction
    - Reduces dispatch overhead, enables inlining
  - Optional *static type annotations*
    - Annotations on variables, parameters, fields enforced dynamically
    - Make specialization more effective

Example of algorithm choice

- Consider solving a matrix equation Ax = b
  - Solution can be expressed as x = A \ b
- Julia has a special type for *Tridiagonal matrices:*

```
In[3]:  strang(n) = SymTridiagonal(2*ones(n),-ones(n-1))
        strang(7)

Out[3]: 7x7 SymTridiagonal{Float64}:
         2.0  -1.0   0.0   0.0   0.0   0.0   0.0
        -1.0   2.0  -1.0   0.0   0.0   0.0   0.0
         0.0  -1.0   2.0  -1.0   0.0   0.0   0.0
         0.0   0.0  -1.0   2.0  -1.0   0.0   0.0
         0.0   0.0   0.0  -1.0   2.0  -1.0   0.0
         0.0   0.0   0.0   0.0  -1.0   2.0  -1.0
         0.0   0.0   0.0   0.0   0.0  -1.0   2.0
```

- Applying the \ operator selects an efficient O(n) impl:

```
In[4]:  strang(8)\ones(8)

Out[4]: 8-element Array{Float64,1}:
         4.0
         7.0
         9.0
        10.0
        10.0
         9.0
         7.0
         4.0
```

Source: Bezanson et al.,
Julia: A Fresh Approach to Numerical Computing. SIAM Review, 2017

Multiple Dispatch

- Ordinary dispatch: choose method based on receiver

  x.multiply(y) // *selects implementation based on class of x*

  - Note: overloading changes this slightly, but relies on static type rather than run-time type
- Multiple dispatch: choose method based on both types

```
⊕(x::Int,     y::Int)      = add(x,y)
⊕(x::Float64,y::Float64) = vaddsd(x,y)
⊕(x::Int,     y::Float64) = vaddsd(vcvtsi2sd(x),y)
⊕(x::Float64,y::Int)      = y ⊕ x
```

Works for Matrices too

```
# Dense + Dense
⊕(A::Matrix, B::Matrix) =
      [A[i,j]+B[i,j] for i in 1:size(A,1),j in 1:size(A,2)]
# Dense + Sparse
⊕(A::Matrix, B::AbstractSparseMatrix) = A ⊕ full(B)
# Sparse + Dense
⊕(A::AbstractSparseMatrix,B::Matrix)  = B ⊕ A # Use Dense + Sparse
# Sparse + Sparse is best written using the long form function definition:
function ⊕(A::AbstractSparseMatrix, B::AbstractSparseMatrix)
    C=copy(A)
    (i,j)=findn(B)
    for k=1:length(i)
        C[i[k],j[k]]+=B[i[k],j[k]]
    end
    return C
end
```

Specialization/Inlining in Julia

```
function vsum(x)
    sum = zero(x)
    for i = 1:length(x)
      @inbounds v = x[i]
      if !is_na(v)
        sum += v
       end
    end
    sum
end
```

```
zero(::Array{T}) where {T<:AbstractFloat} = 0.0
zero(::Array{T}) where {T<:Complex} = complex(0.0,0.0)
zero(x) = 0

is_na(x::T) where T = x == typemin(T)

typemin(::Type{Complex{T}}) where {T<:Real}
      = Complex{T}(-NaN)
```

Source: Bezanson et al., Julia: Dynamism and Performance Reconciled by Design. PACMPL(OOPSLA) 2018.

Specialization/Inlining in Julia

```
function vsum(x)
    sum = zero(x)
    for i = 1:length(x)
      @inbounds v = x[i]
      if !is_na(v)
        sum += v
       end
    end
    sum
end
```

```
zero(::Array{T}) where {T<:AbstractFloat} =
zero(::Array{T}) where {T<:Complex} = comple
zero(x) = 0

is_na(x::T) where T = x == typemin(T)

typemin(::Type{Complex{T}}) where {T<:Real}
      = Complex{T}(-NaN)
```

Resulting assembly same as C

```
        push    %rbp
        mov     %rsp, %rbp
        mov    (%rdi), %rcx
        mov     8(%rdi), %rdx
        xor     %eax, %eax
        test    %rdx, %rdx
        cmove   %rax, %rdx
        movl    $1, %esi
        movabs  $0x8000000000000000, %r8
        jmp     L54
        nopw    %cs:(%rax,%rax)
L48:    add     %rdi, %rax
        inc     %rsi
L54:    dec     %rsi
        nopl    (%rax)
L64:    cmp     %rsi, %rdx
        je      L83
        mov     (%rcx,%rsi,8), %rdi
        inc     %rsi
        cmp     %r8, %rdi
        je      L64
        jmp     L48
L83:    pop     %rbp
        ret
        nopw    %cs:(%rax,%rax)
```

Source: Bezanson et al., Julia: Dynamism and Performance Reconciled by Design. PACMPL(OOPSLA) 2018.

Type Inference

```
function f(a,b)
  c = a+b
  d = c/2.0
  return d
end
```

```
function f(a::Int,b::Int)
  c = a+b::Int
  d = c/2.0::Float64
  return d
end => Float64
```

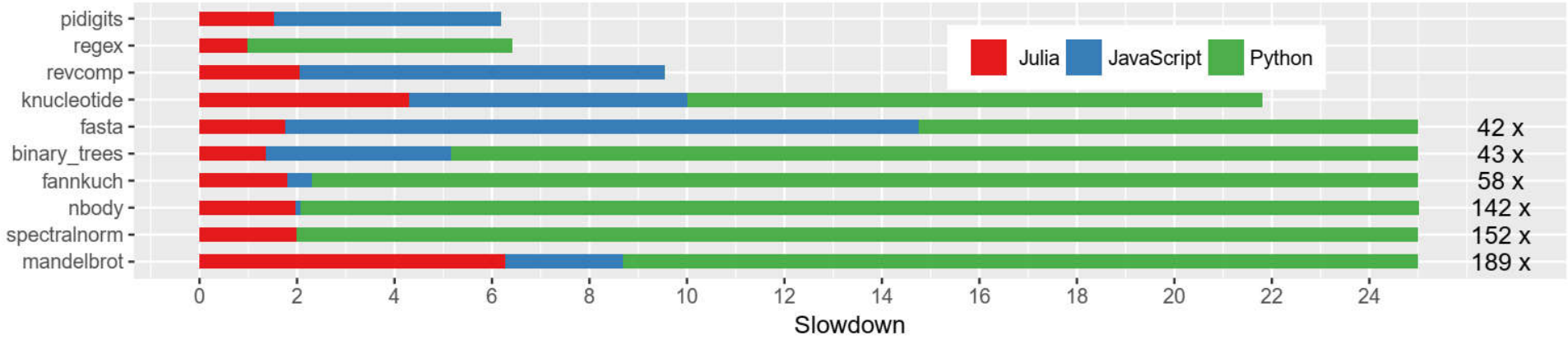
Interprocedural

```
function a()
  return b(3)+1
end
function b(num)
  return num+2
end
```

```
function a()
  return b(3)+1::Int
end => Int
function b(num::Int)
  return num+2::Int
end => Int
```

Source: Bezanson et al., Julia: Dynamism and Performance Reconciled by Design. PACMPL(OOPSLA) 2018.

Does it Work?



Remaining performance loss mostly due to memory operations (e.g. GC)
Outliers: regex just calls C implementation; knucleotide written for clarity over performance; mandelbrot lacks vectorization

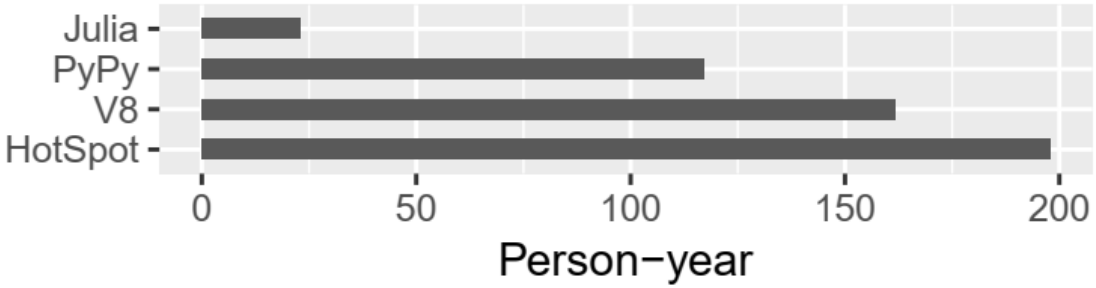


Fig. 6. Time spent on implementations

Source: Bezanson et al., Julia: Dynamism and Performance Reconciled by Design. PACMPL(OOPSLA) 2018.

Why Good Performance in Julia

…despite so little effort on the implementation?

- Dispatch & specialization
  - Chooses right algorithm based on run-time types
  - Specialize implementation for actual run-time types encountered
    - Allows inlining, unboxing, further optimization
- Programmer discipline
  - Type annotations on fields
    - Allows compiler to infer types read from the heap
    - It knows types of arguments from dispatch/specialization
  - Type stability
    - Code is written so that knowing the concrete types of arguments allows the compiler to infer concrete types for all variables in the function
      - Thus specialized code becomes monomorphic: no dispatch, no polymorphism
    - Maintained by programmer discipline

Zero Cost Abstraction: From C to C++

- Starts with C, but adds abstraction facilities (and a little dynamism)
- Motto: "Zero-cost abstraction"
  - C++ can perform similarly to C, but is (somewhat) higher-level
- Generic programming: Static specialization with templates
  - Templated code is parameterized by one or more types T
  - A copy is generated for each instantiation with a concrete type
  - Can get genericity with static dispatch instead of dynamic
    - Same benefits as Julia, no GC overhead (unless you choose to add it)
    - More language complexity, and little more safety than C

Adding Safety in C++

- Memory issues one of the big problems in C, early C++
- Modern solution: smart pointers
  - The pointer itself is an abstraction
  - Method calls are passed on to the object

```
unique_ptr<Obj> p(new Obj());
unique_ptr<Obj> q = move(p);
q->foo(); // OK
p->foo(); // illegal; the pointer is in q now

// deallocate q's memory automatically when q goes out of scope
```

Adding Safety in C++

- Memory issues one of the big problems in C, early C++
- Modern solution: smart pointers
  - The pointer itself is an abstraction
  - Method calls are passed on to the object

```
shared_ptr<Obj> p(new Obj());
shared_ptr<Obj> q = p; // reference count increments
q->foo(); // OK
p->foo(); // OK

// deallocate memory automatically when both p and q go out of scope
```

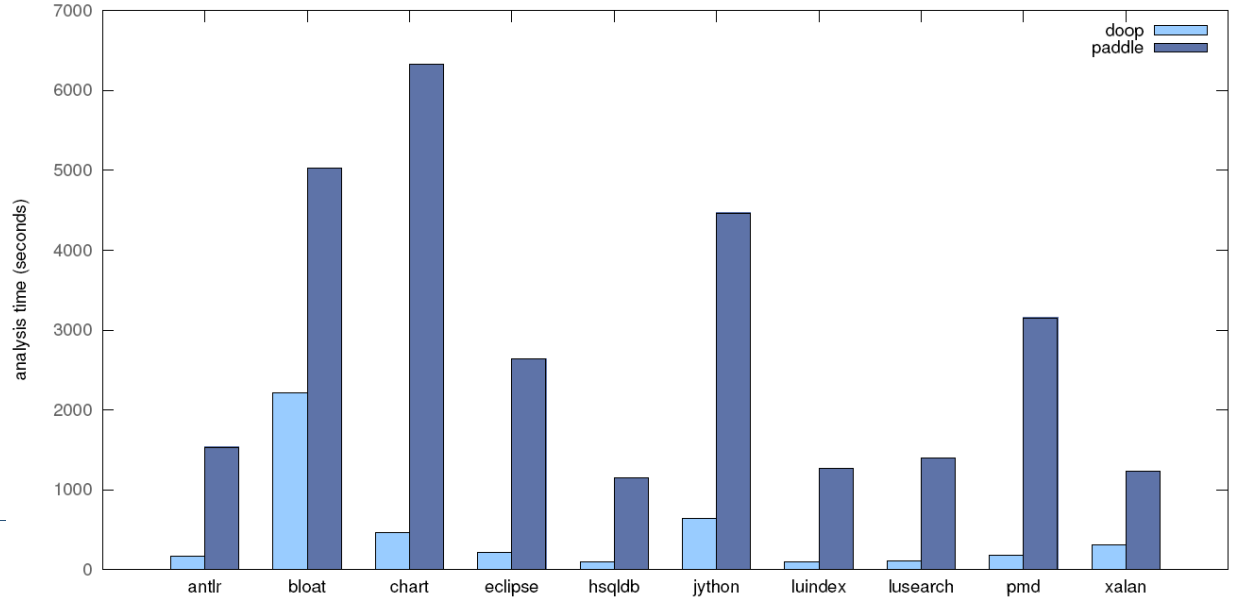
Modern C++ programming is completely different from when I taught the language circa 2001 due to smart pointers

Rust: Ownership Types for Memory Safety

- Rust keeps "close to the metal" like C, provides abstraction like C++
- Safety achieved via ownership types
  - Like in Obsidian, every block of memory has an owner
  - Adds power using regions
    - A region is a group of objects with the same lifetime
    - Allocated/freed in LIFO (stack) order
    - Younger objects can point to older ones
    - Type system tracks region of each object, which regions are younger
  - Fast and powerful—but (anecdotally) hard to learn
    - Nevertheless anyone in this class could do it!
- Unsafe blocks allow bending the rules
  - But clients see a safe interface

Domain-Specific Paths to Performance

- Domain-Specific Language
  - Captures a particular program domain
  - Usually restricted – sometimes not Turing-complete
  - Execution strategy takes advantage of domain restrictions
- Examples
  - DataLog – bottom-up logic programming
    - Dramatic performance enhancements on problems like alias analysis



Domain-Specific Paths to Performance

- Domain-Specific Language
  - Captures a particular program domain
  - Usually restricted – sometimes not Turing-complete
  - Execution strategy takes advantage of domain restrictions
- Examples
  - DataLog – bottom-up logic programming
    - Dramatic performance enhancements on problems like alias analysis
    - Infers new facts from other known facts until all facts are generated
    - Optimization based on database indexing controlled by programmer
  - SAT/SMT solving – logical formulas
    - Based on DPLL and many subsequent algorithmic improvements
  - SPIRAL (a CMU project!)
    - Optimization of computational kernels across platforms
    - Like Fortran parallelization, but with more declarative programs and auto-tuning for the platform

Datalog Examples

- See separate presentation on Declarative Static Program Analysis with Doop, slides 1-4, 18-30, 34-37, and 62-68

http://www.cs.cmu.edu/~aldrich/courses/17-355-18sp/notes/slides20-declarative.pdf

Summary

- Tradeoff between performance and abstraction/safety/dynamism
- Approaches to this tradeoff
  - Giving programmers control (Fortran, C, C++)
  - Smart dynamic compilers (Java, JavaScript, Python, etc.)
  - Smart parallelization (Fortran, C)
  - Compiler assumptions + programmer discipline (Fortran)
  - Good libraries (Fortran, C, Julia, Python)
  - Abstraction and generic programming (C++)
  - Types for memory safety (Rust)
  - Multiple dispatch + specialization + programmer discipline (Julia)
  - Domain-specific languages and optimizations (DataLog, SPIRAL, SAT/SMT solvers)