Julia: Dynamism and Performance Reconciled by Design

JEFF BEZANSON, Julia Computing, Inc.
JIAHAO CHEN, Capital One
BEN CHUNG, Northeastern University
STEFAN KARPINSKI, Julia Computing, Inc
VIRAL B. SHAH, Julia Computing, Inc
LIONEL ZOUBRITZKY, École Normale Supérieure and Northeastern University
JAN VITEK, Northeastern University and Czech Technical University

Julia is a programming language for the scientific community that combines features of productivity languages, such as Python or MATLAB, with characteristics of performance-oriented languages, such as C++ or Fortran. Julia's productivity features include: dynamic typing, automatic memory management, rich type annotations, and multiple dispatch. At the same time, it allows programmers to control memory layout and leverages a specializing just-in-time compiler to eliminate much of the overhead of those features. This paper details the design choices made by the creators of Julia and reflects on the implications of those choices for performance and usability.

CCS Concepts: • **Software and its engineering** → **Language features**; *General programming languages*; *Just-in-time compilers*; Multiparadigm languages;

Additional Key Words and Phrases: multiple dispatch, just in time compilation, dynamic languages

ACM Reference Format:
Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Lionel Zoubritzky, and Jan Vitek. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 00 (2018), 23 pages. https://doi.org/00.0000/0000000

1 INTRODUCTION

Scientific programming has traditionally adopted one of two programming language families: productivity languages (Python, MATLAB, R) for easy development, and performance languages (C, C++, Fortran) for speed and a predictable mapping to hardware. Features of productivity languages such as dynamic typing or garbage collection make exploratory and iterative development simple. Thus, scientific applications often begin their lives in a productivity language. In many cases, as the problem size and complexity outgrows what the initial implementation can handle, programmers turn to performance languages. While this usually leads to improved performance, converting an existing application (or some subset thereof) to a different language requires significant programmer involvement; features previously handled by the language (e.g. memory management) now have to be emulated by hand. As a result, porting software from a high level to a low level language is often a daunting task.

Scientists have been trying to bridge the divide between performance and productivity for years. One example is the ROOT data processing framework [Antcheva et al. 2015]. Confronted with petabytes of data, the high energy physics community spent more than 20 years developing an extension to C++, providing interpretive execution and reflection—typical features of productivity

2475-1421/2018/00-ART00
https://doi.org/00.0000/0000000

languages—while retaining C++'s performance in critical portions of the code. Most scientific fields, however, do not have the resources to build and maintain their own computational infrastructure.

The Julia programming language aims to decrease the gap between productivity and performance languages. On one hand, it provides productivity features like dynamic typing, garbage collection, and multiple dispatch. On the other, it has a type-specializing just-in-time compiler and lets programmers control the layout of data structure in memory. Julia, therefore, promises scientific programmers the ease of a productivity language at the speed of a performance language.

This promise is surprising. Dynamic languages like Python or R typically suffer from at least an order of magnitude slowdown over C, and often more. Fig. 1 illustrates that Julia is indeed a dynamic language. It declares a `Node` datatype containing two untyped fields, `val` and `nxt`, and an untyped `insert` function that takes a sorted list and performs an ordered insertion. While this code will be optimized by the Julia compiler, it is not going to run at full speed without some additional programmer intervention.

```
mutable struct Node
  val
  nxt
end

function insert(list, elem)
  if list isa Void
    return Node(elem, nothing)
  elseif list.val > elem
    return Node(elem, list)
  end
  list.nxt = insert(list.nxt, elem)
  list
end
```

Fig. 1. Linked list

The key to performance in Julia lies in the synergy between language design, implementation techniques and programming style. Julia's design was carefully tailored so that a very small team of language implementers could create an efficient compiler. The key to this relative ease is to leverage the combination of language features and programming idioms to reduce overhead, but what language properties enable easy compilation to fast code?

Language design: Julia includes a number of features that are common to many productivity languages, namely dynamic types, optional type annotations, reflection, dynamic code loading, and garbage collection. A slightly less common feature is symmetric multiple dispatch [Bobrow et al. 1986]. In Julia a function can have multiple implementations, called methods, distinguished by the type annotations added to parameters of the function. At run-time, a function call is dispatched to the most specific method applicable to the types of the arguments. Julia's type annotations can be attached to datatype declarations as well, in which case they are checked whenever typed fields are assigned to. Julia differentiate between concrete and abstract types: the former can be instantiated while the latter can be extended by subtypes. This distinction is important for optimization.

Language implementation: Performance does not arise from great feats of compiler engineering: Julia's implementation is simpler than that of many dynamic languages. The Julia compiler has three main optimizations that are performed on a high-level intermediate representation; native code generation is delegated to the LLVM compiler infrastructure. The optimizations are (1) *method inlining* which devirtualizes multi-dispatched calls and inline the call target; (2) *object unboxing* to avoid heap allocation; and (3) *method specialization* where code is special cased to its actual argument types. The compiler does not support the kind of speculative compilation and deoptimizations common in dynamic language implementations, but supports dynamic code loading from the interpreter and with `eval()`.

The synergy between language design and implementation is in evidence in the interaction between the three optimizations. Each call to a function that has a combination of concrete argument types not observed before triggers specialization. A type inference algorithm uses the type of the arguments (and if these are user-defined types, the declared type of fields) to discover the types of variables in the specialized function. This enables both unboxing and inlining. The specialized method is added to the function's dispatch table so that future calls with the same argument types can use the generated code.

Programming style: To assist the implementation, Julia programmers need to write idiomatic code that can be compiled effectively. Programmers are keenly aware of the optimizations that the compiler performs and write code that is shaped accordingly. For instance, adding type annotations to fields of datatypes is viewed as good practice. Another good practice is to write methods that are *type stable*. A method is type stable if, when it is specialized to a set of concrete types, type inference can assign concrete types to all variables in the function. This property should hold for all specializations of the same method. Type instability can stem from methods that can return values of different types, or from assignment of different types to the same variable depending on branches of the function.

This paper gives the first unified overview of the design of the language and its implementation, paying particular attention to the features that play a role in achieving performance. This furthers the work of Bezanson et al. [2017] by detailing the synergies at work through the entire compilation pipeline between the design and the programming style of the language. Moreover we present experimental results on performance and usage. More specifically, we give results obtained on a benchmark suite of 10 small applications where Julia v0.6.2 performs between 0.9x and 6.1x from optimized C code. On our benchmarks, Julia outperforms JavaScript and Python in all cases. Finally we conduct a corpus analysis over a group of 50 popular projects hosted on GitHub to examine how the features and underlying design choices of the language are used in practice. The corpus analysis confirms that multiple dispatch and type annotations are widely used by Julia programmers. It also shows that the corpus is mostly made up of type stable code.

2 JULIA IN ACTION

To introduce Julia, we consider an example. This code started as an attempt to replicate the R language's multi-dimensional summary function. This shortened version computes the sum of a vector. Just like the R function, the Julia code is polymorphic over vectors of integer, float, boolean, and complex values. Furthermore, since R supports missing values in every data type, we encode `NA` s in Julia.[1]

Fig. 2 shows how to sum values. The syntax is straightforward. In this case, type annotations are not needed for the compiler to optimize the code. Variables are lexically scoped; an initial assignment defines them. Fig. 3 is the output of `@code_native(vsum([1]))`, printing the x86 machine code for `vsum([1])`. It is noteworthy that the generated machine code does not contain object allocation or method invocation, nor does it invoke any language runtime components. The machine code is similar to code one would expect to be emitted by a C compiler.

Type stability is key to performant Julia code, allowing the compiler to optimize using types. An expression is type stable if, in a given type context, it always returns a value of the same type. Function `vsum(x)` always returns a value that is either of the same type as the element type of `x` (for floating point and complex vectors) or `Int64`. For the call `vsum([1])`, the method returns an `Int64`, as its argument is of type `Array{Int64,1}`. When presented with such a call, the Julia compiler specializes the method for that type. Specialization provides enough information to determine that all values manipulated by the computation are of the same type, `Int64`. Thus, no boxing is required; moreover, all calls are devirtualized and inlined. The `@inbounds` macro elides array bounds checking.

Type stability may require cooperation from the developer. Consider variable `sum`: its type has to match the element type of `x`. In our case, `sum` must be appropriately initialized to support any of the possible argument types integer, float, complex or boolean. To ensure type stability, the programmer leverages dispatch and specialization with the definition of the function `zero` shown

[1] Julia v0.7 adds support for missing values.

in Fig. 4. It dispatches on the type of its argument. If the argument is an array containing subtypes of float, the function returns float `0.0`. Similarly, if passed an array containing complex numbers, the function returns a complex zero. In all other cases, it returns integer 0. All three methods are trivially type stable, as they always return the same value for the same types.

Missing values also require attention. Each primitive type needs its own representation—yet the code for checking whether a value is missing must remain type stable. This can be achieved by leveraging dispatch. We add a function `is_na(x)` that returns true if `x` is missing. We select the smallest value in each type to use as its missing value (obtained by calling `typemin`).

The solution outlined so far fails for booleans, as their minimum is `false`, which we can't steal. Fig. 5 shows how to add a new boolean data type, `RBool`. Like Julia's boolean, `RBool` is represented as an 8-bit value; but like R's boolean, it has three values. Defining a new data type entails providing a constructor and a conversion function. Since our data type has only three useful values, we enumerate them as constants. We add a method to `typemin` to return `NA`. Finally, since the loop adds booleans to integers, we need to extend addition to integer and `RBool`.

```
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
```

Fig. 2. Compute vector sum

```
        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)
```

Fig. 3. `@code_native vsum([1])` (X86-64)

```
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)
```

Fig. 4. `zero` yields the zero matching the element type, by default the integer `0`. `is_na` checks for missing values encoded as the smallest element of a type (returned by the builtin function `typemin`). `typemin` is extended with a method to return the smallest complex value

```
primitive type RBool 8 end

RBool(x::UInt8) = reinterpret(RBool, x)
convert(::Type{T},x::RBool) where{T<:Real}
    = T(reinterpret(UInt8,x))

const T  = RBool(0x1)
const F  = RBool(0x0)
const NA = RBool(0xff)

typemin(::Type{RBool}) = NA

+(x::Union{Int,RBool}, y::RBool) = Int(x) + Int(y)
```

Fig. 5. `RBool` is a an 8-bit primitive type representing boolean values extended with a missing (`NA`. The constructor takes an 8-bit unsigned integer. Conversion allows to cast any number into an `RBool`. A new method is added to `typemin` to return `NA`

3 EVALUATING RELATIVE PERFORMANCE

Julia has to be fast to compete against other languages used for scientific computing. Competitors like C, C++ and Fortran offer speed but require greater expertise from programmers; others like Python, R, and MATLAB offer high-level abstractions at the expense of speed. Julia strives for a compromise. This goal is a difficult one, however, as dynamic languages are notoriously difficult to optimize. One additional issue to consider is that of language development time. Fig. 6 shows the person-years invested in several implementations. These rough measures were obtained using the projects' commit histories: two commits made by the same developer in one week were counted as one person-week of effort. This figure suggests that performance comes at a substantial cost in engineering. For example, V8 for Javascript and HotSpot for Java have nearly two centuries invested into their respective implementations. Even PyPy, an academic project, has over one century of work. Given the difference in implementation effort, Julia's performance is surprising.

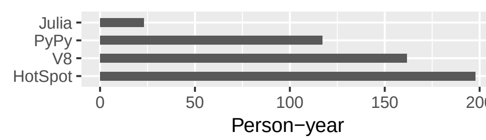

Fig. 6. Time spent on implementations

To estimate the languages' relative performance, we selected 10 small programs for which implementations in C, JavaScript, and Python are available in the programming language benchmark game (PLBG) suite [Gouy 2018]. The PLBG suite consists of small but non-trivial benchmarks which stress either computational or memory performance. We started with PLBG programs from the Julia team, and fixed several performance anomalies. The benchmarks are written in an idiomatic style, using the same algorithms as the C benchmarks. Their code is largely untyped, with type annotations only appearing on structure fields. Over the 10 benchmark programs, 12 type annotations appear, all on structs and only in the nbody, binary_trees, and knucleotide. The `@inbounds` macro eliding bounds checking is the only low-level optimization used, leveraged only in revcomp. Using the PLBG methodology, we measured the size of the programs by removing comments and duplicate whitespace characters, then performing the minimal GZip compression. The combined size of all the benchmarks is 6.0 KB for Julia, 7.4 KB for JavaScript, 8.3 KB for Python and 14.2 KB for C.

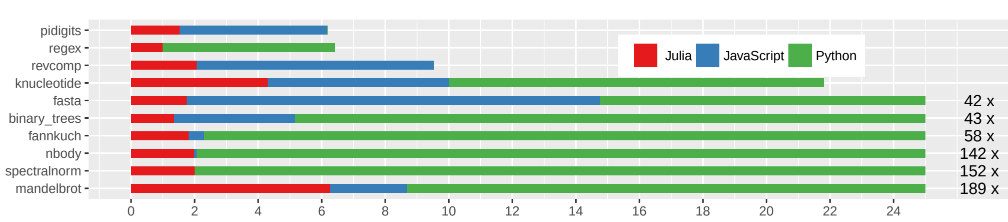

Fig. 7. Slowdown of Julia, JavaScript, and Python relative to C

Fig. 7 compares the performance of the four languages with the results normalized to the running time of the C programs. Measurements were obtained using Julia v0.6.2, CPython 3.5.3, V8/Node.js v8.11.1, and GCC 6.3.0 -O2 for C, running on Debian 9.4 on a Intel i7-950 at 3.07GHz with 10GB of RAM. All benchmarks ran single threaded. No other optimization flags were used.

The results show Julia consistently outperforming Python and Javascript (with the exception of spectralnorm). Julia is mostly within 2x of C. Slowdowns are likely due to memory operations. Like other high level dynamically-typed programming languages, Julia relies on a garbage collector to manage memory. It prohibits the kind of explicit memory management tricks that C allows. In particular, it allocates structs on the heap. Stack allocation is only used in limited circumstances. Moreover, Julia disallows pointer arithmetic.

Three programs fall outside of this range: two programs (knucleotide and mandelbrot) have slowdowns greater than 2x over C, while one (regex) is faster than C. The knucleotide benchmark was written for clarity over performance; it makes heavy use of abstractly-typed struct fields (which cause the values they denote to be boxed). In the case of mandelbrot, the C code is manually vectorized to compute the fractal image 8 pixels at a time; Julia's implementation, however, computes one pixel at a time. Finally, regex, which was within the margin of error of C, simply calls into the same regex library C does.

Julia is fast on tiny benchmarks, but this may not be representative of real-world programs. We lack the benchmarks to gauge Julia's performance at scale. Some libraries have published comparisons. JuMP, a large embedded domain specific language for mathematical optimization, is one such library. JuMP converts numerous problem types (e.g. linear, integer linear, convex, and nonlinear) into standard form for solvers. When compared to equivalent implementations in C++, MATLAB, and Python, JuMP is within 2x of C++. For comparison, MATLAB libraries are between 4x and 18x slower than C++, while Python's optimization frameworks are at least 70x slower than C++ [Lubin and Dunning 2013]. This provides some evidence that Julia's performance on small benchmarks may be retained for larger programs.

4 THE JULIA PROGRAMMING LANGUAGE

The designers of Julia set out to develop a language specifically for the needs of scientific computation, and they chose a finely tuned set of features to support this use case. Antecedent languages, like R and MATLAB, illustrate scientific programmers' desire to write high-level scripts, which motivated Julia's adoption of an optionally typed surface language. Likewise, these languages drove home the importance of flexibility: programmers regularly extend the core languages' functionalities to fit their needs. Julia provides this extensibility mechanism through multiple dispatch.

4.1 Values, types, and annotations

4.1.1 Values. Values can be either instances of *primitive types*, represented as sequences of bits, or *composite types*, represented as a collection of fields holding values. Logically, every value is tagged by its full type description; in practice, however, tags are often elided when they can be inferred from context. Composite types are immutable by default, thus assignment to their fields is not allowed. This restriction is lifted when the `mutable` keyword is used.

4.1.2 Types declarations. Programmers can declare three kinds of types: *abstract types*, *primitive types*, and *composite types*. Types can be parametrized by bounded type variables and have a single supertype. The type `Any` is the root of the type hierarchy, or the greatest supertype (top). *Abstract* types cannot be instantiated; *concrete* types can.

```
abstract type Number end

abstract type Real <: Number end

primitive type Int64 <: Signed 64 end

struct Polar{T<:Real} <: Number
  r::T
  t::T
end
```

The code shown is an extract of Julia's numeric tower. `Number` is an abstract type with no declared supertype, which means `Any` is its super type. `Real` is also abstract but has `Number` as its super type. `Int64` is a primitive type with `Signed` as its supertype; it is represented in 64 bits. The struct `Polar{T<:Real}` is a subtype of `Number` with two fields of type `T` bounded by `Real`. Run-time checks ensure that values stored in these fields are of the declared type. When types are omitted from field declarations, fields can hold values of `Any` type. Julia does not make a distinction between reference and value types as Java does. Concrete types can be manipulated either by value or by reference; the choice is left to the implementation. Abstract types, however, are always manipulated by reference. It is noteworthy that composite types do not admit subtypes; therefore, types such as `Polar` are final and cannot be extended with additional fields.

4.1.3 Type annotations. Julia offers a rich type annotation language to express constraints on fields, parameters, local variables, and method return types. The `::` operator ascribes a type to a definition. The annotation language includes union types, written `Union{A,...}`; tuple types, written `Tuple{A,...}`; iterated union types, written `TExp where A<:T<:B`; and singleton types, written `Type{T}` or `Val{V}`. The distinguished type `Union{}`, with no argument, has no value and acts as the bottom type.

Union types are abstract types which include, as values, all instances of their arguments. Thus, `Union{Integer,String}` denotes the set of all integers and strings. Tuple types describe the types of the elements that may be instantiated within a given tuple, along with their order. They are parametrized, immutable types. Additionally, they are *covariant* in their parameters. The last parameter of a tuple type may optionally be the special type `Vararg`, which denotes any number of trailing elements.

Julia provides iterated union types to allow quantification over a range of possible instantiations. For example, the denotation of a polar coordinate represented using a subtype `T` of real numbers is `Polar{T} where Union{}<:T<:Real`. Each **where** clause introduces a single type variable. The type application syntax `T{A}` requires `T` to be a **where** type, and substitutes `A` for the outermost type variable in `T`. Type variable bounds can refer to outer type variables. For example,

```
Tuple{T,Array{S}} where S<:AbstractArray{T} where T<:Real
```

refers to 2-tuples whose first element is some `Real`, and whose second element is an `Array` whose element type is the type of the first tuple element, `T`.

A singleton type is a special kind of abstract type, `Type{T}`, whose only instance is the object `T`.

4.1.4 Subtyping. In Julia, the subtyping relation between types, written `<:`, is used in run-time casts, as well as method dispatch. Semantic subtyping partially influenced Julia's subtyping [Frisch et al. 2002], but practical considerations caused Julia to evolve in a unique direction. Julia has an original combination of *nominal subtyping*, *union types*, *iterated union types*, *covariant* and *invariant* constructors, and *singleton types*, as well as the *diagonal rule*. Parametric types are invariant in their parameters because of Julia's memory representation of values. Arrays of dissimilar values box each of their arguments, for consistent element size, under type `Array{Any}`. However, if all the values are statically determined to be of the same kind, they are stored inside of the array itself because their memory layout is known. Tuple types represent both tuples and function arguments. They are covariant because of this latter use, which allows Julia to compute dispatch using subtyping of tuples. Subtyping of union types is asymmetrical but intuitive. Whenever a union type appears on the left-hand side of a judgment, as in `Union{T1,...} <: T`, all the types `T1,...` must be subtypes of `T`. In contrast, if a union type appears on the right-hand side, as in `T <:Union{T1,...}`, then only one type, `Ti`, needs to be a supertype of `T`. Covariant tuples are distributive with respect to unions, so `Tuple{Union{A,B},C}<:Union{Tuple{A,C},Tuple{B,C}}`. The iterated union construct `Texp where A<:T<:B`, as with union types, must have either a forall or an exist semantics, according to whether the union appears on the left or right of a subtyping judgment. Finally, the *diagonal rule* states that if a variable occurs more than once in covariant position, it is restricted to ranging over only concrete types. For example, `Tuple{T,T} where T` can be seen as `Union{Tuple{Int8,Int8},Tuple{Int16,Int16},...}`, where `T` ranges over all concrete types. The details of the subtyping algorithm are intricate and the interactions between its features can be surprising, we describe those in a companion paper [Nardelli et al. 2018].

4.1.5 Dynamically-checked type assertions. Type annotations in method arguments are guaranteed by the language semantics. However, Julia allows the insertion of type annotations elsewhere in the program that act as type *assertions*. For example, to guarantee that variable `x` has type `Int64` in the remainder of the program, the type assertion `x::Int64 =...` can be inserted into its declaration. Likewise, functions can assert a return type: the function `f()::Int =...` is one example. Composite type fields can also be annotated. These type annotations check the type of the expression's or field's value. If it is not a subtype of the declared type, Julia will try to convert it to the declared type. If this conversion fails, Julia will throw an exception. As a result, while these type annotations look like those in statically typed languages, their semantics are slightly different.

4.2 Multiple dispatch

Julia uses multiple dispatch extensively, allowing extension of functionality by means of overloading. Multiple dispatch uses every argument type to figure out the target of each function call. In Julia, each function (for example +) can consist of an arbitrarily large number of methods (in the case of +, 180). Each of these methods declares what types it can handle, and Julia will dispatch to whichever method is most specific for a given function call. As hinted at with addition, multiple dispatch is omnipresent. Virtually every operation in Julia involves dispatch. New methods can then be added to existing functions, extending them to work with new types.

```
struct Dual{T}
  re::T
  dx::T
end

function Base.:(+)(a::Dual{T},b::Dual{T}) where T
  Dual{T}(a.re+b.re, a.dx+b.dx)
end
function Base.:(*)(a::Dual{T},b::Dual{T}) where T
  Dual{T}(a.re*b.re, a.dx*b.re+b.dx*a.re)
end
function Base.:(/)(a::Dual{T},b::Dual{T}) where T
  Dual{T}(a.re/b.re, (a.dx*b.re-a.re*b.dx)/(b.re*b.re))
end
```

4.2.1 Example. Libraries can add their own implementations of basic math operators. For example, forward differentiation is a technique that allows derivatives to be calculated for arbitrary programs. It is implemented by passing both a value and its derivative through a program. In many languages, the code being differentiated would have to be aware of forward differentiation as the dual numbers need new definitions of arithmetic. Multiple dispatch allows to implement a library that works for existing functions, as we can simply extend arithmetic operators. Suppose we want to compute the derivative of `f(a,b)=a*b/(b+b*a+b*b)` about `a`, with `a =1` and `b =3`. By overloading arithmetic, the same operators found in `f` work on dual numbers; thus, taking the derivative of `f` is as simple as calling `f` with dual numbers: `f(Dual(1.0,1.0),Dual(3.0,0.0)).dx` yields `0.16`.

4.2.2 Semantics. Dispatching on a function `f` for a call with argument type T consists in picking a method m from all the methods of `f`. The selection filters out methods whose types are not a supertype of T and takes the method whose type T' is the most specific of the remaining ones. In contrast to single dispatch, every position in the tuples T and T' have the same role—there is no single receiver position that takes precedence. Specificity is required to disambiguate between two or more methods which are all supertypes of the argument type. It extends subtyping with extra rules, allowing comparison of dissimilar types as well. The specificity rules are defined by the implementation and lack a formal semantics. In general, `A` is more specific than `B` if `A != B` and either `A <: B` or one of a number of special cases hold:

(a) `A = T{P}` and `B = S{Q}`, and there exist values of `P` and `Q` such that `T <: S`. This allows us to conclude that `Array{T} where T` is more specific than `AbstractArray{String}`.
(b) Let `C` be the non-empty meet (approximate intersection) of `A` and `B`, and `C` is more specific than `B` and `B` is not more specific than `A`. This is used for union types: `Union{Int32,String}` is more specific than `Number` because the meet, `Int32`, is clearly more specific than `Number`.
(c) `A` and `B` are tuple types, `A` ends with a `Vararg` type and `A` would be more specific than `B` if its `Vararg` was expanded to give it the same number of elements as `B`. This tells us that `Tuple{Int32,Vararg{Int32}}` is more specific than `Tuple{Number,Int32,Int32}`.
(d) `A` and `B` have parameters and compatible structures, `A` provides a consistent assignment of non-`Any` types to replace `B`'s type variables, regardless of the diagonal rule. This means that `Tuple{Int,Number,Number}` is more specific than `Tuple{T,S,S} where {T,S<:T}`.
(e) `A` and `B` have parameters and compatible structures and `A`'s parameters are equal or more specific than `B`'s. As a consequence, `Tuple{Array{T} where T,Number}` is more specific than `Tuple{AbstractArray{String},Number}`.

One more interesting feature is dispatch on type objects and on primitive values. For example, the Base library's `nutple` function is defined as a set of methods dispatching on the value of their second argument. Thus a call to `ntuple(id,Val{2})` yields `(1,2)` where `id` is the identity function. The `@_inline_meta` macro is used to force inling.

```
ntuple(f, ::Type{Val{0}}) = (@_inline_meta; ())
ntuple(f, ::Type{Val{1}}) = (@_inline_meta; (f(1),))
ntuple(f, ::Type{Val{2}}) = (@_inline_meta; (f(1), f(2)))
```

4.3 Metaprogramming

Julia provides various features for defining functions at compile-time and run-time and has a particular definition of visibility for these definitions.

4.3.1 Macros. Macros provide a way to generate code in the final body of a program. The use of macros is intended to reduce the need for calls to `eval()` that are so frequent in other dynamic languages. A macro maps a tuple of arguments to an expression which is compiled directly. Macro arguments may include expressions, literal values, and symbols. The example on the right shows the definition of the `assert` macro which either returns `nothing` if the assertion is true or throws an exception with an optional message provided by the user. The `:(...)` syntax denotes quotation, that is the creation of an expression. Within it, values can be interpolated: `$x` will be replaced by the value of `x` in the expression.

```
macro assert(ex, msgs...)
  msg_body = isempty(msgs) ? ex : msgs[1]
  msg = string(msg_body)
  return :($ex ? nothing
     : throw(AssertionError($msg)))
end
```

4.3.2 Reflection. Julia provides methods for run-time introspection. The names of fields may be interrogated using `fieldnames()`. The type of each field is stored in the `types` field of each composite value. Types are themselves represented as a structure called `DataType`. The direct subtypes of any `DataType` may be listed using `subtypes()`. The internal representation of a `DataType` is important when interfacing with C code and several functions are available to inspect these details. `isbits(T::DataType)` returns true if `T` is stored with C-compatible alignment. The builtin function `fieldoffset(T::DataType,i::Integer)` returns the offset for field `i` relative to the start of the type. The methods of any function may be listed using `methods()`. The method dispatch table may be searched for methods accepting a given type using `methodswith()`.

More powerful is the `eval()` function which takes an expression object and evaluates it in the global scope of the current module. For example `eval(:(1+2))` will take the expression `:(1+2)` and evaluate it yielding the expected result. When combined with an invocation to the parser, any arbitrary string can be evaluated, so for instance `eval(parse("function id(x) x end"))` adds an identity method. One important difference from languages such as JavaScript is that `eval()` does not have access to the current scope. This is crucial for optimizations as it means that local variables are protected from interference. The `eval()` function is sometimes used as part of code generation. Here for example is a generalization of some of the basic binary operators to three arguments. This generates four new methods of three arguments each.

```
for op in (:+, :*, :&, :|)
  eval(:($op(a,b,c) = $op($op(a,b),c)))
end
```

4.3.3 Epochs. The Julia implementation keeps a world age (or *epoch*) counter. The epoch is a monotonically increasing value that can be associated to each method definition. When a method is compiled, it is assigned a minimum world age, set to the current epoch, and a maximum world age, set to `typemax(Int)`. By default, a method is visible only if the current epoch is superior to its minimum world age. This prevents method redefinitions (through `eval` for instance) from affecting the scope of currently invoked methods. However, when a method is redefined, the maximum world age of all its callers gets capped at the current epoch. This in turn triggers a lazy recompilation of the callers at their next invocation. As a consequence, a method always invokes its callee with its latest version defined at the compile time of the caller. When the absolute latest (independent of compile epoch) version of a function is needed, programmers can use `Base.invokelatest(fun,args)` to bypass this mechanism; however, these calls cannot be statically optimized by the compiler.

4.4 Discussion

The design of Julia makes a number of compromises, and we discuss some of the implications here.

Object oriented programming. Julia's design does not support the class-based object oriented programming style familiar from Java. Julia lacks the encapsulation that is the default in languages going back all the way to Smalltalk: all fields of a struct are public and can be accessed freely. Moreover, there is no way to extend a point class `Pt` with a color field as one would in Java; in Julia the user must plan ahead for extension and provide a class `AbsPt`. Each "class" in that programming style is a pair of an abstract and a concrete class. One can define methods that work on abstract classes such as the `move` method which takes any point and new coordinates. The `copy` methods are specific to each concrete "class" as they must create instances. As first discussed by Chung and Li [2017], the unfortunate side effect of the untyped nature of Julia and of the fact that abstract classes have neither fields nor methods is that there is no documentation to remind the programmer that a `copy` method is needed for `ColPt`. This has to be discovered by inspection of the code.

```
abstract type AbsPt end
struct Pt <: AbsPt
  x::Int
  y::Int
end

abstract type AbsColPt <: AbsPt end
struct ColPt <: AbsColPt
  x::Int
  y::Int
  c::String
end

copy(p::Pt, dx, dy) = Pt(p.x+dx, p.y+dy)
copy(p::ColPt, dx, dy) =
   ColPt(p.x+dx, p.y+dy, p.c)

move(p::AbsPt, dx, dy) = copy(p, dx, dy)
```

Functional programming. Julia supports several functional programming idioms—higher order functions, immutable-by-default values—but has no arrow types. Instead, the language ascribes functions to have incomparable nominal types without argument or return type information. Thus, many traditional typed idioms are impractical, and it is impossible to dispatch on function types. However, nominal types do allow dispatch on methods passed as arguments, enabling a different set of patterns. For example, the implementation of `reduce` delegates to a special-purpose function `reduce_empty` which, given a function and list type, determines the value corresponding to the empty list. If reducing with `+`, the natural empty reduction value is 0, for the correct 0. Capturing this, `reduce_empty` has the following definition: `reduce_empty(::typeof(+),T)=zero(T)` . In this case, `reduce_empty` dispatches the nominal `+` function type, then returns the zero element for `T`.

Gradual typing. The goal of gradual type systems is to allow dynamically typed programs to be extended with type annotations after the fact [Siek 2006; Tobin-Hochstadt and Felleisen 2006]. Julia's type system superficially appears to fit the bill; programs can start untyped, and, step by step, end up fully decorated with type annotations. But there is a fundamental difference. In a gradually typed language, a call to a function `f(t::T)`, such as `f(x)`, will be statically checked to ensure that the variable `x`'s declared type matches the argument's type `T`. In Julia, on the other hand, a call to `f(x)` will not be checked statically; if `x` does not have type `T`, then Julia throws a runtime error. Another difference is that, in Julia, a variable, parameter, or field annotated with type `T` will *always* hold a value of type `T`. Gradual type systems only guarantee that values will act like type `T`, wrapping untyped values with contracts to ensure they they are indistinguishable [Tobin-Hochstadt and Felleisen 2008]. If a gradually-typed program manipulates a value erroneously, that error will be flagged and blame will be assigned to the part of the program that failed to respect the declared types. Similarly, Julia departs from optional type systems, like Hack [Facebook 2016] or Typescript [Microsoft 2016]. These optional type systems provide no guarantee whatsoever about what values a variable of type `T` actually holds. Julia is closest in spirit to Thorn [Bloom et al. 2009]. The two languages share a nominal subtype system with tag checks on field assignment and method calls. In both systems, a variable of type `T` will only ever have values of type `T`. However, Julia differs substantially from Thorn, as it lacks a static type system and adds multiple dispatch.

5 IMPLEMENTING JULIA

Julia is engineered to generate efficient native code at run-time. The Julia v0.6.2 compiler is an optimizing just-in-time compiler structured in three phases: source code is first parsed into abstract syntax trees; those trees are then lowered into an intermediate representation that is used for Julia level optimizations; once those optimizations are complete, the code is translated into LLVM IR and machine code is generated by LLVM [Lattner and Adve 2004].

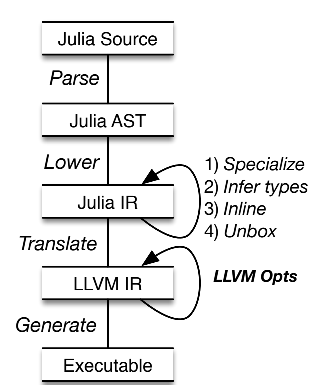

Fig. 8. Julia JIT compiler

Fig. 8 is a high level overview of the compiler pipeline. With the exception of the standard library which is pre-compiled, all Julia code executed by a running program is compiled on demand. The compiler is relatively simple: it is a method-based JIT without compilation tiers; once methods are compiled they are not changed as Julia does not support deoptimization with on-stack replacement.

Memory is managed by a stop-the-world, non-moving, mark-and-sweep garbage collector. The mark phase can be executed in parallel. The collector has a single old generation for objects that survive a number of cycles. It uses a shadow stack to record pointers in order to be precise.

Since v0.5, Julia natively supports multi-threading but the feature is still labeled as "experimental". Parallel loops use the `Threads.@threads` macro which annotates **for** loops that are to run in a multi-threaded region. Other part of the multi-threaded API are still in flux. An alternative to Julia native threading is the ParallelAccelerator system of Anderson et al. [2017] which generates OpenMP code on the fly for parallel kernels. The system crucially depends on type stability—code that is not type stable will execute single threaded.

Fig. 9 gives an overview of the implementation of Julia v0.6.2. The standard library, Core, Base and a few other modules, accounts for most of the use of Julia in Julia's implementation. The middle-end is written in C; C++ is used for LLVM IR code generation. Finally, Scheme and Lisp are used for the front end. External dependencies such as LLVM, which is used as back end, do not participate to this figure.

Language	files	code
Julia	296	115,252
C	79	44,930
C++	21	18,491
Scheme	11	6,369
C/C++ Header	44	6,205
Lisp	6	1,901
make	7	684
Bourne Shell	2	85
Assembly	4	74
	470	193,991

Fig. 9. Source files

5.1 Method specialization

Julia's compilation strategy is built around runtime type information. Both type inference and JIT compilation will happen every time a method is called with a new tuple of argument types. In effect, by dynamically inferring function types, Julia is able to take a dynamically typed program and monomophize it. Every time a method is called with a new tuple of argument types, it is specialized by the runtime for these types. Optimizing methods at invocation time, rather than ahead of time, provides the JIT with key pieces of information: the memory layout of all arguments are known, allowing for unboxing and direct field access.

Devirtualization is the process of replacing virtual function invocations with invocations of a single specialization. Dispatching on an argument is a needless effort if it can be statically shown that the argument will only ever have a single type; directly calling the method specialized for the known type is much more efficient. As a result, devirtualization can reduce dispatch overhead and enable inlining, discussed later.

This compilation process is rather slow (due to LLVM), however, and its results are cached. Once compiled, method specializations are never discarded. As a result, methods are only compiled the first time they are called with a new type. The next time it is called with the same type, the specialized version is called instead and execution is fast. This process converges quickly as long as functions are only ever called with a limited number of types. Type stability ensures this condition.

Compiling for every new function argument does create a new pathology. If a function gets called only a few times under each of many argument type tuples, then virtually every invocation will incur the substantial cost of specialization. The language runtime cannot solve every instance of this problem, as programs that generate an infinite number of new call signatures, an extreme version of type instability, are easy to write. However, Julia makes several decisions that simplify the process of writing type stable functions in practical applications.

The biggest real issue would arise from argument with potentially many types. Julia allows tuple types to contain a `Vararg` component, which can be expanded infinitely. To avoid unbounded numbers of specializations for functions with varargs arguments, Julia treats these as having type `Any`. Likewise, in Julia, each function value has its own type, so methods that take function arguments could exhibit this pathology. Yet, specializing over the type of the function can be useful in order to inline it. The selected heuristic consists in specializing if the function is called in the method body, and treating it as having type `Any` otherwise. Other heuristics are involved for arguments having type `Type` for similar reasons.

The language runtime has little recourse for type unstable code beyond generating a large number of specializations. As a last resort, Julia (0.7) programmers can use the `@nospecialize` annotation to prevent specialization on a specific argument if they expect it to not be type stable.

5.2 Type inference

Type inference enables many of Julia's key optimizations. It runs after specializing and is used for inlining and unboxing. Julia uses a set constraint based type inference system with constraints arising from return values, method dereferences, and argument types. Type requirements need to be satisfied at function call sites and field assignments. The system propagates constraints forward to satisfy requirements, inferring the types for intermediate values along the way.

Given the concrete types of all function arguments, intraprocedural type inference propagates types forward into the method body. An example is shown in Fig. 10. When `f` is called with a pair of integers, type inference finds that `a+b` returns an integer; therefore `c` is likewise an integer. From this, it follows that `d` is a float and so is the return type of the method. Note that this explanation relies on knowing the return type of `+`. Since addition could be overloaded, it is necessary to be able to infer the return types of arbitrary methods. Return types may vary depending on argument type, and previous inference results may not cover the current case. Therefore, when a new function is called, inference on the caller must be suspended and continue on the called function to figure out the return type of the call.

```
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
```

Fig. 10. A simple example of type inference

Interprocedural type inference is simple for non-recursive methods as seen in Fig. 11: abstract execution flows to the called method and the return type is computed. For recursive methods cycle elimination is performed. Once a cycle is identified, it is executed within the abstract interpreter until it reaches convergence. The cycle is then contracted into a single monolithic function from the perspective of type inference. More challenging are methods whose argument or return types can grow indefinitely depending on its arguments. To avoid this, Julia limits the size of the inferred types to an arbitrary bound. In this manner, the set of possible types is finite and therefore termination of the abstract interpretation system is guaranteed.[2]

```
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
```

Fig. 11. Simple interprocedural type inference

5.3 Method inlining

Inlining replaces a function call by the body of the called function. In Julia, it can be realized in a very efficient way because of its synergy with specialization and type inference. Indeed, if the body of a method is type stable, then the internal calls can be inlined. Conversely, inlining can help type inference because it gives additional context. For instance, inlined code can avoid branches that can be eliminated as dead code, which allows in turn to propagate more precise type information. Yet, the memory cost incurred by inlining can be sometimes prohibitive; moreover it requires additional compilation time. As a consequence, inlining is bounded by a number of pragmatic heuristics.

5.4 Object unboxing

Since Julia is dynamic, a variable may hold values of many types. As a consequence, in the general case, values are boxed, allocated on the heap with a tag that specifies their type. Unboxing is the optimization that consists in manipulating values directly. This optimization is made possible by

[2] In Julia v0.7 this limitation is replaced by a more complex heuristic to determine whether the type is growing in a call cycle.

a combination of design choices. First, since concrete types are final, a concrete type specifies both the size of a value and its layout. This would not be the case in Java (due to subtyping) or TypeScript (due to structural subtyping). In addition, Julia does not have a null value; if it did, there would be need for an extra tag. As a consequence, values of plain data types can always be stored unboxed. Repeated boxing and unboxing can be expensive Unboxing can also be impossible to realize although the type information is present, in particular for recursive data structures. As with inlining, heuristics are thus used to determine when to perform this optimization.

6 JULIA IN PRACTICE

In order to understand how programmers use the language, we analyzed a corpus of 50 packages hosted on GitHub. Choosing packages over programs was a necessity: no central repository exists of Julia programs. Packages were included based on GitHub stars. Selected packages also had to pass their own test suites. Additionally, we analyzed Julia's standard library.

6.1 Typeful programming

Julia is a language where types are entirely optional. Yet, knowing them is highly profitable at compile time since it enables major optimizations. Users are thus encouraged to program in a typeful style where code is, as much as possible, type stable. To what extent is this rule followed?

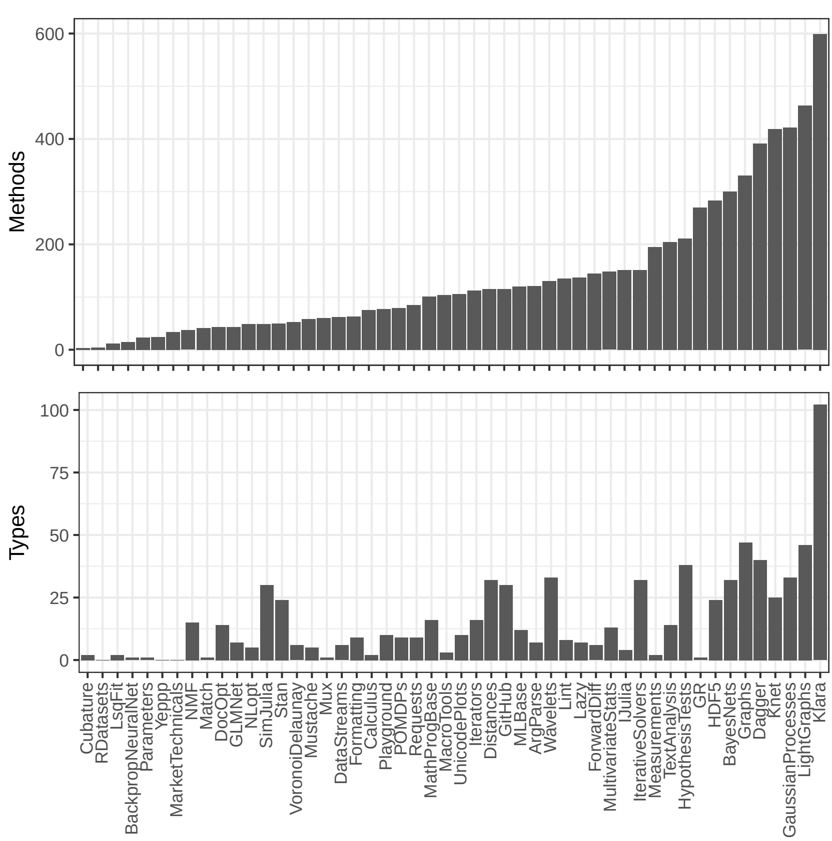

Fig. 12. Number of methods and types by package

6.1.1 Type annotations. Fig. 12 gives the number of methods and types defined in each packages (excluding Base). We analyzed our corpus after it was loaded into Julia to ensure that generated methods could be captured; we preformed structural analysis of parsed ASTs, allowing us to measure only methods and types written by human developers. In total, the corpus includes 792 type definitions and 7,018 methods. The median number of types and methods per package is 9 and 104, respectively. Klara, a library for Markov chain Monte Carlo inference, is the largest package by both number of types and methods with 102 and 599, respectively. Three packages, MarketTechnicals, RDatasets, and Yeppp, define zero types; while Cubature defines just 3 methods, the fewest in the corpus. Clearly, Julia users define many types and functions. However, the level of dynamism remains a question.

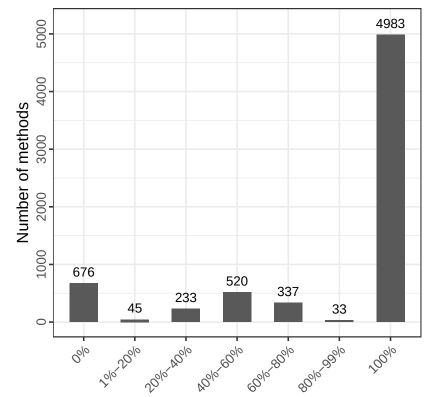

Fig. 13. Methods by percentage of typed arguments

Fig. 13 shows the distribution of type annotations on arguments of method definitions. 0% means all arguments are untyped (`Any`), while 100% means that all arguments are non-`Any`. An impressive 4,983 (or 62%) of methods are fully type-annotated.

Despite having the opportunity to write untyped methods, library developers define mostly typed methods and only a few untyped and partially typed methods. This behavior may not reflect that of the average user of Julia, though, because library developers are biased toward writing optimized code; and in Julia, this requires precisely controlling the types.

6.1.2 Type stability. Type inference can only attribute concrete types to variables if these variables can be statically determined to be of that type. Type stability is key to devirtualizing methods and inlining them as well as to unboxing. We capture the presence of type instability at run time by dynamic analysis on the test suites of our corpus. Each function call was recorded, along with the tuple of types of its arguments, the called method, and the call site. We filtered out calls to anonymous and compiler-generated functions to focus on functions defined by humans.

Fig. 14 compares, for each package, the number of call sites where all the calls targeted only one specialized method to those that call two and more. The y-axis is shown in log scale. On average, 92% of call sites target a single specialized method. Code is thus in general type stable, which agrees with the assumption that programmers attempt to write type stable code.

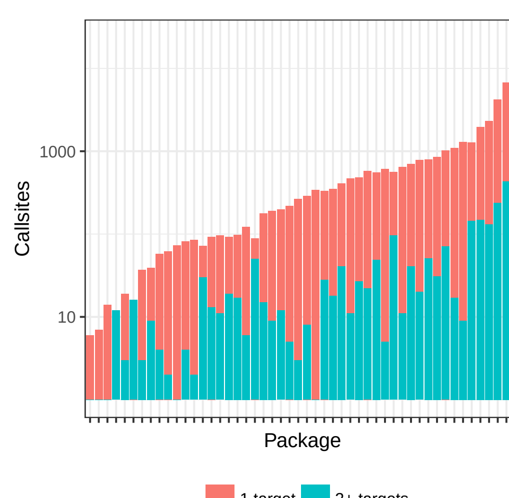

Fig. 14. Targets per callsite per package

6.2 Multiple dispatch

Multiple dispatch is the most prominent features of Julia's design. Its synergy with specialization is crucial to understand the performance of the language and its ability to inline devirtualize and inline efficiently. Thus, how is multiple dispatch used from a programmer's perspective? Moreover, a promise of multiple dispatch is that it can be used to extend existing behavior with new implementations. Two questions arise: how much do Julia libraries extend existing functionality, and what functionality do they extend?

6.2.1 Dispatch metrics. The standard means by which to compare usage of multiple dispatch is by Muschevici et al. [2008]. We focus on the dispatch ratio (DR), the number of methods defined per function, and the degree of dispatch (DoD), the average number of argument positions needed to select a concrete method. These metrics are computed statically, and have no dynamic component.

Fig. 15 compares Julia to other languages with multiple dispatch, using data from Muschevici et al. [2008]. The data for Julia was collected on all the functions exported by the Base library. Julia shows the highest value of dispatch ratio with an average of almost 5 methods defined per function. This is in part due to the presence of a small number of functions with an extremely high number of overloads: `convert` for instance, which is used to to convert a value to a given type, has 699 overloads.

Language	Functions	DR	DoD
Dylan (OpenDylan)	2143	2.51	0.39
CLOS (McCLIM)	2222	2.43	0.78
Cecil (Vortex)	6541	2.33	0.36
Diesel (Whirlwind)	5737	2.07	0.32
Nice (NiceC)	1184	1.36	0.15
Julia	1292	4.89	0.85

Fig. 15. Muschevici et al. metrics

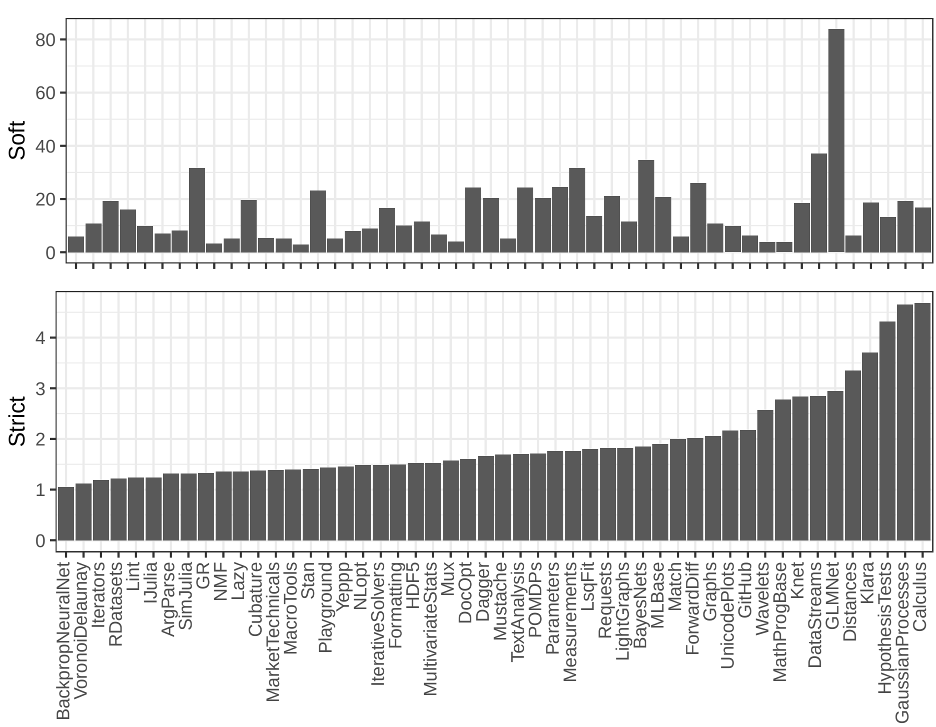

Fig. 16. Dispatch ratio with soft vs. strict elimination

But even without those outliers, 73% of the functions have at least two methods, which demonstrates the importance of overloading in the standard library. The degree of dispatch is also the highest, which shows that the full extent of multiple dispatch is used, and not only overloading which is a consequence of it.

These metrics assume a single monolithic code base. However, when comparing multiple programs that import shared libraries (e.g. Core and Base), the question is how to avoid double counting libraries? Since methods of the same function can be defined in different packages, which methods should be kept? Two answers are possible. First, every method reachable from an imported module, and which belongs to a function having at least one method defined in the target package. We call this "soft elimination," as it precludes definitions unreachable from the package, but includes some imported definitions. Second, we could say that only functions that have all their methods defined within the target package package count. We call this "strict elimination."

Fig. 16 shows the dispatch ratio across our corpus using soft and strict elimination. Despite being nominally the same metric, the dispatch ratios are not correlated. At issue is the nature of imports. If a package overloads + with a single new method, then strict elimination will not count it. However, soft elimination will count it along with the 180 methods from the standard library. If the package under consideration only has a few functions, its dispatch ratio could be greater than 20—four times higher than the maximum observed with strict elimination—despite its small size.

Figure 17 gives the total number of arguments dispatched on per function. It is the cumulative result of all the package after strict elimination, ensuring that no function has been duplicated. The functions without any argument were filtered out because of their trivial dispatch. 79% of the functions can still be dispatched on 0 argument, which shows that the arity of the function is in most of the cases enough to determine the corresponding method.

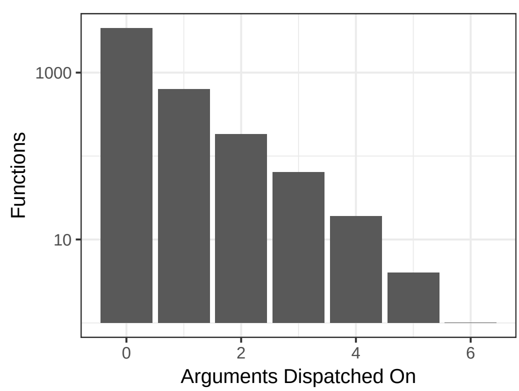

Fig. 17. Number of arguments by dispatched on

6.2.2 Overloading. Fig. 18 examines how multiple dispatch is used to extend existing functionality. We use the term *external overloading* to mean that a package adds a method to a function defined in a library. Packages are binned based on the percentage of functions that they overload versus define. Packages with only external overloading are at 100%, while packages that do not use external overloading would be in the 0% bin. Many packages are defined without extensive use of external overloading. For 28 out of 50 packages, fewer than 30% of the functions they define are overloads. However, the distribution of overloading has a long tail, with a few libraries relying on overloads heavily. The Measurements package has the highest proportion of overloads, with 147 overloads out of a total of 161 methods (91%). This is justified by the purpose of Measurements: it propagates errors throughout other operations, which is done by extending existing functions.

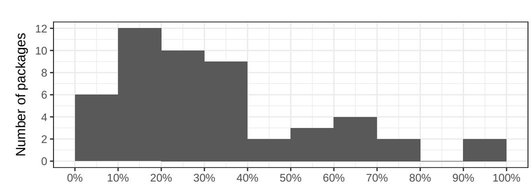

Fig. 18. Packages by % of overloaded functions

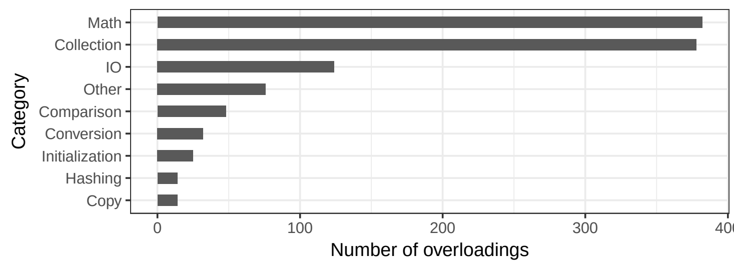

Fig. 19. Function overloads by category

To address the question of what is overloaded, we manually categorized the top 20th quantile of overloaded functions (128 out of 641) into 9 groups. Fig. 19 depicts how many times functions from each group is overloaded. Multiple dispatch is used heavily to overload mathematical operators, like addition or trigonometric functions. Libraries overload existing operators to work with their own types, providing natural interfaces and interoperability with existing code. Examples include Calculus, which overloads arithmetic to allow symbolic expressions; and ForwardDiff, which can compute numerical derivatives of existing code using dual numbers that act just like normal values. Collection functions also are widely overloaded. Many libraries have collection-like objects, and by overloading these methods they can use their collections where Julia expects any abstract collection. However, Julia's interfaces are only defined by documentation, as a result of its dynamic design. The `AbstractArray` interface can be extended by any struct, and it is only suggested in the documentation that implementations should overload the appropriate methods. Use cases for math and collection extension are easy to come by, so their prevalence is unsurprising. However, the lack of overloads in other categories illustrates some surprising points. For example, the large number of IO, math, and collection overloads (which implement variations on `tostring`) suggest a preponderance of new types. However, few overloads to compare, convert, or copy are provided.

6.3 Specializations

Figure 20 gives the number of specializations per method recorded dynamically on our corpus. The data uses strict eliminations, so that the results from different packages can be summed without

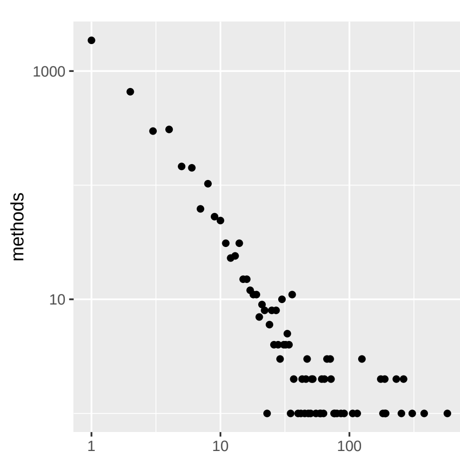

Fig. 20. Number of specializations per method

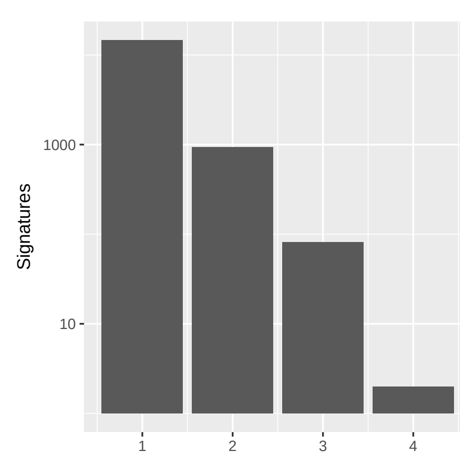

Fig. 21. Applicable methods per call signature

duplicate functions. The distribution has a heavy tail, which shows that programmers actually write methods that can be very polymorphic. Note that polymorphism is not in contradiction with type stability, since a method called with different tuples of argument types across different call sites can be type stable for each of its call sites. Conversely, 46% of the methods have only been specialized once after running the tests. Many methods are thus used monomorphically: this hints that a number of methods may have a type specification that prevent polymorphism, which means that programmers tend to think of the concrete types they want their methods applied to, rather than only an abstract type specification.

Figure 21 corroborates this hypothesis. It represents the number of applicable methods per call signature. A method is applicable if the tuple of types corresponding to the requirements for its arguments is a supertype of that of the actual call. This data is collected on dynamic traces for functions with at least two methods. 93% of the signatures can only dispatch to one method, which strongly suggests that methods tend to be written for disjoint type signatures. As a consequence it shows that the specificity rules, used to determine which method to call, boil down to subtyping in the vast majority of cases.

6.4 Impact on performance

Fig. 22 illustrates the impact on performance of LLVM optimizations, type inference and devirtualization. By default Julia uses LLVM at optimization level `O2`. Switching off all LLVM optimizations generates code between 1.1x and 7.1x slower. Turning off type inference means that method are specialized correctly but all internal operations will be performed on values of type `Any`. Functions that have only a single method may still be devirtualized and dispatched to. The graph is capped at 100x slowdown. The actual slowdowns range between 5.6x and 2151x. Lastly, turning off devirtualization implies that no inlining will be performed and all function calls are dispatched dynamically. The slowdowns range between 5.3x and 1905x.

Obviously, Julia was designed to be optimized with type information. These results suggest that performance of fully dynamic code is rather bad. It is likely that if users were to write more dynamic code, some of the techniques that have proved successful for other dynamic languages could be ported to Julia. But clearly, the current implementation crucially relies on code being type stable and on devirtualization and inlining. The impact of the LLVM optimizations is small in comparison.

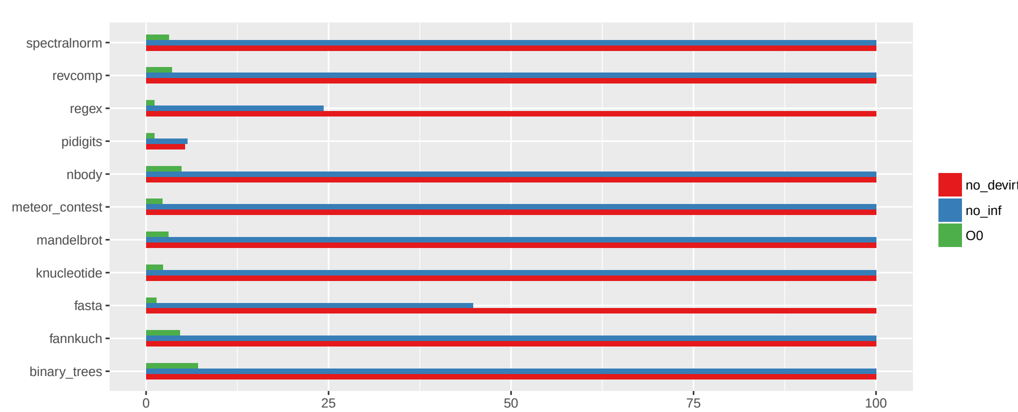

Fig. 22. Optimization and performance

7 RELATED WORK

Julia occupies an interesting position in the programming language landscape. We review some of the related work and compare the most relevant work to Julia.

Scientific computing languages. R [R Core Team 2008] and MATLAB [MATLAB 2018] are the two languages superficially closest to Julia. Both languages are dynamically typed, garbage collected, vectorized and offer an integrated development environment focused on a read-eval-print loop. However, the languages' attitudes towards vectorization differ. In R and MATLAB, vectorized functions are more efficient than iterative code whereas the contrary stands for Julia. In this context we use "vectorization" to refer to code that operates on entire vectors[3], so for instance in R, all operations are implicitly vectorized. The reason vectorized operations are faster in R and MATLAB is that the implicit loop they denote is written in a C library, while source-level loops are interpreted and slow. In comparison, Julia can compile loops very efficiently, as long as type information is present.

While there has been much research in compilation of R [Kalibera et al. 2014; Talbot et al. 2012; Würthinger et al. 2013] and MATLAB [Chevalier-Boisvert et al. 2010; De Rose and Padua 1999], both languages are far from matching the performance of Julia. The main difference, in terms of performance, between MATLAB or R, and Julia comes from language design decisions. MATLAB and R are more dynamic than Julia, allowing, for example, reflective operations to inspect and modify the current scope and arbitrary redefinition of functions. Other issues include the lack of type annotations on data declarations.

Other languages have targeted the scientific computing space, most notably IBM's X10 [Charles et al. 2005] and Oracle's Fortress [Steele et al. 2011]. The two languages are both statically typed, but differ in their details. X10 focuses on programming for multicore machines that have partitioned global addressed spaces; its type system is designed to track the locations of values. Fortress, on the other hand, had multiple dispatch like Julia, but never reached a stage where its performance could be evaluated due to the complexity of its type system. In comparison, Julia's multi-threading is still in its infancy, and it does not have any support for partitioned address spaces.

Multiple dispatch. Multiple dispatch goes back to Bobrow et al. [1986] and is used in languages such as CLOS [DeMichiel and Gabriel 1987], Perl [Randal et al. 2003] and R [Chambers 2014]. Lifting explicit programmatic type tests into dispatch requires an expressive annotation sublanguage to capture the same logic; expressiveness that has created substantial research challenges. Researchers have struggled with how to provide expressiveness while ensuring type soundness. Languages such as Cecil [Litvinov 1998] and Fortress [Allen et al. 2011] are notable for their rich type systems; but, as mentioned in Guy Steele's retrospective talk, finding an efficient, expressive and sound type system remains an open challenge.[4] The language design trade-off seems to be that programmers want to express relations between arguments that require complex types, but when types are rich enough, static type checking becomes difficult. The Fortress designers were not able to prove soundness, and the project ended before they could get external validation of their design. Julia side-steps many of the problems encountered in previous work on typed programming languages with multiple dispatch. It makes no attempt to statically ensure invocation soundness or prevent ambiguities, falling back to dynamic errors in these cases.

Static type inference. At heart, despite the allure of types and the optimizations they allow, type inference for untyped programs is difficult. Flow typing tries to propagate types through

[3]This discussion should not be confused with hardware-level vectorization, e.g. SIMD operations, which are available to Julia at the LLVM level.

[4]JuliaCon 2016, https://www.youtube.com/watch?v=EZD3Scuv02g.

the program at large, but sacrifices soundness in the process. Soft typing [Fagan 1991] applies Hindley-Milner type inference to untyped programs, enabling optimizations. This approach has been applied practically in Chez Scheme [Wright and Cartwright 1994]. However, Hindley-Milner type inference is too slow to use on practically large code bases. Moreover, many language features (such as subtyping) are incompatible with it. Constraint propagation or dataflow type inference systems are a commonly used alternative to Hindley-Milner inference. These systems work by propagating types in a data flow analysis [Aiken and Wimmers 1993]. No unification is needed, and it is therefore much faster and more flexible than soft typing. Several inference systems based on data flow have been proposed for JavaScript [Chaudhuri et al. 2017], Scheme [Shivers 1990], and others.

Dynamic type inference for JIT optimizations. Feeding dynamic type information into a type propagation type inference system is not a technique new to Julia. The first system to use dataflow type inference inside a JIT compiler was RATA [Logozzo and Venter 2010]. RATA relies on abstract interpretation of dynamically-discovered intervals, kinds, and variations to infer extremely precise types for Javascript code; types which enable JIT optimizations. The same approach was then used in 2012 by Hackett and Guo [2012], which used a simplified type propagation system to infer types for more general Javascript code, providing performance improvements. In comparison to dynamic type inference systems for Javascript, Julia's richer type annotations and multiple dispatch allow it to infer more precise types. Another related project is the StaDyn [Garcia et al. 2016] language. StaDyn was designed specifically with hybrid static and dynamic type inference in mind. However, StaDyn does not have many of Julia's features that enable precise type inference, including typed fields and multiple dispatch.

Dynamic language implementation. Modern dynamic language implementation techniques can be traced back to the work of Hölzle and Ungar [1994] on the Self language, who pioneered the ideas of run-time specialization and deoptimization. These ideas were then transferred into the Java HotSpot compiler [Paleczny et al. 2001]; in HotSpot, static type information can be used to determine out object layout, and deoptimization is used when inlining decisions were invalidated by newly loaded code. Implementations of JavaScript have increased the degree of specialization, for instance allowing unboxed primitive arrays at the more complex guards and potentially wide-ranging deoptimization [Würthinger et al. 2013].

Other Julia papers. The Julia team's paper [Bezanson et al. 2017] differs from the present paper in that it is more introductory in nature, targeting the scientific community. It does not discuss the implementation of the language, nor does it perform a corpus analysis. The performance figures reported in the earlier work are for micro-benchmarks only.

8 CONCLUSION

This paper has argued that productivity and performance can be reconciled. Julia is a language for scientific computing that offers many of the features of productivity languages, namely rapid development cycles; exploratory programming without having to worry about types or memory management; reflective and meta-programming; and language extensibility via multiple dispatch. In spite of these features, however, a relatively simple language implementation yields speed competitive with that of performance languages.

The language implementation is a just in time compiler which performs three main optimizations: method specialization, method inlining and object unboxing. Code generation is delegated to the LLVM infrastructure. The Julia implementation avoids the complex speculation and deoptimization games played in other dynamic languages by using the concept of world age, a time stamp on compiled code, to trigger recompilation.

The language design is tailored to these optimizations. Multiple dispatch means that any function call, by default, looks up the most applicable method given the type of the arguments; thus any method specialization can be made immediately accessible to the entire program by simply extending the dispatch table for the corresponding function. The ability to annotate data structure declarations with types is helpful to the compiler as the type and the layout of fields can be specified. Combined with the restriction on subtyping concrete types—and the absence of nulls—this facilitates unboxing. The limits on reflection allow type inference to be more precise than in similar dynamic languages, which, in turn, makes inlining and unboxing more successful.

Finally, the programming style adopted by Julia users favors type stable functions. These functions are easier to optimize as they are written so that every variable can be assigned a single concrete type during method specialization. To achieve this, programmers replace branches on types by generic calls and push all of their type testing logic into the multiple dispatch mechanism.

While our observations are encouraging, Julia is still a young language. More experience is needed to draw definitive conclusions as most programs are small and written by domain experts. How the approach we describe here will scale to large (multi-million lines long) programs and to domains outside of scientific computing is a question we hope to answer in future work.

Acknowledgments. This work received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (award 1544542 and award 1518844) as well as ONR (award 503353).

REFERENCES

Alexander Aiken and Edward L Wimmers. 1993. Type inclusion constraints and type inference. In *Proceedings of the conference on Functional programming languages and computer architecture*. ACM, 31–41.

Eric Allen, Justin Hilburn, Scott Kilpatrick, Victor Luchangco, Sukyoung Ryu, David Chase, and Guy Steele. 2011. Type Checking Modular Multiple Dispatch with Parametric Polymorphism and Multiple Inheritance. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2048066.2048140

Todd A. Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. 2017. Parallelizing Julia with a Non-Invasive DSL. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.4

Ilka Antcheva, Maarten Ballintijn, Bertrand Bellenot, Marek Biskup, Rene Brun, Nenad Buncic, Philippe Canal, Diego Casadei, Olivier Couet, Valery Fine, Leandro Franco, Gerardo Ganis, Andrei Gheata, David González Maline, Masaharu Goto, Jan Iwaszkiewicz, Anna Kreshuk, Diego Marcos Segura, Richard Maunder, Lorenzo Moneta, Axel Naumann, Eddy Offermann, Valeriy Onuchin, Suzanne Panacek, Fons Rademakers, Paul Russo, and Matevz Tadel. 2015. ROOT - A C++ Framework for Petabyte Data Storage, Statistical Analysis and Visualization. *CoRR* abs/1508.07749 (2015). http://arxiv.org/abs/1508.07749

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017). https://doi.org/10.1137/141000671

Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. 2009. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/1639950.1640016

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. 1986. CommonLoops: Merging Lisp and Object-oriented Programming. In *Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/28697.28700

John Chambers. 2014. Object-Oriented Programming, Functional Programming and R. *Statist. Sci.* 2 (2014). Issue 29. https://doi.org/10.1214/13-STS452

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. (2005). https://doi.org/10.1145/1103845.1094852

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 48.

Maxime Chevalier-Boisvert, Laurie J. Hendren, and Clark Verbrugge. 2010. Optimizing Matlab through Just-In-Time Specialization. In *Conference on Compiler Construction (CC)*. 46–65. https://doi.org/10.1007/978-3-642-11970-5_4

Benjamin Chung and Paley Li. 2017. Towards Typing Julia. In *The -2th Workshop on New Object-Oriented Languages (NOOL)*.

Luiz De Rose and David Padua. 1999. Techniques for the Translation of MATLAB Programs into Fortran 90. *ACM Trans. Program. Lang. Syst.* 21, 2 (March 1999). https://doi.org/10.1145/316686.316693

Linda DeMichiel and Richard Gabriel. 1987. The Common Lisp Object System: An Overview. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/3-540-47891-4_15

Facebook. 2016. Hack. (2016). http://hacklang.org.

Mike Fagan. 1991. *Soft typing: an approach to type checking for dynamically typed languages.* Ph.D. Dissertation. Rice University.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic Subtyping. In *Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1109/LICS.2002.1029823

Miguel Garcia, Francisco Ortin, and Jose Quiroga. 2016. Design and Implementation of an Efficient Hybrid Dynamic and Static Typing Language. *Softw. Pract. Exper.* 46, 2 (Feb. 2016), 199–226. https://doi.org/10.1002/spe.2291

Isaac Gouy. 2018. The Computer Language Benchmarks Game. (2018). https://benchmarksgame-team.pages.debian.net/benchmarksgame

Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. *ACM SIGPLAN Notices* 47, 6 (2012), 239–250.

Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Conference on Programming Language Design and Implementation (PLDIO)*. https://doi.org/10.1145/773473.178478

T. Kalibera, P. Maj, F. Morandat, and J. Vitek. 2014. A Fast Abstract Syntax Tree Interpreter for R. In *Conference on Virtual Execution Environments (VEE)*.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Symposium on Code Generation and Optimization ((CGO)*. https://doi.org/10.1109/CGO.2004.1281665

Vassily Litvinov. 1998. Constraint-based Polymorphism in Cecil: Towards a Practical and Static Type System. In *Addendum to Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA Addendum)*. https://doi.org/10.1145/346852.346948

Francesco Logozzo and Herman Venter. 2010. RATA: rapid atomic type analysis by abstract interpretation–application to javascript optimization. In *International Conference on Compiler Construction*. Springer, 66–83.

Miles Lubin and Iain Dunning. 2013. Computing in Operations Research using Julia. In *INFORMS Journal on Computing*. https://doi.org/10.1287/ijoc.2014.0623

MATLAB. 2018. *version 9.4*. The MathWorks Inc., Natick, Massachusetts.

Microsoft. 2016. TypeScript – Language Specification. (2016).

Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. 2008. Multiple Dispatch in Practice. In *Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/1449764.1449808

Francesco Zappa Nardelli, Artem Pelenitsyn, Julia Belyakova, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. In *Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Symposium on Java Virtual Machine Research and Technology (JVM)*. http://dl.acm.org/citation.cfm?id=1267847.1267848

R Core Team. 2008. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing. http://www.R-project.org

Allison Randal, Dan Sugalski, and Leopold Toetsch. 2003. *Perl 6 and Parrot Essentials.* O'Reilly.

Olin Shivers. 1990. Data-flow Analysis and Type Recovery in Scheme. In *Topics in Advanced Language Implementation*. MIT Press, 47–88.

Jeremy Siek. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. http://ecee.colorado.edu/~siek/pubs/pubs/2006/siek06_gradual.pdf.

Guy Steele, Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. 2011. Fortress (Sun HPCS Language). In *Encyclopedia of Parallel Computing*. 718–735. https://doi.org/10.1007/978-0-387-09766-4_190

Justin Talbot, Zachary DeVito, and Pat Hanrahan. 2012. Riposte: a trace-driven compiler and parallel VM for vector code in R. In *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *Symposium on Dynamic languages (DLS)*. https://doi.org/10.1145/1176617.1176755

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed Scheme. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1328438.1328486

Andrew K Wright and Robert Cartwright. 1994. A practical soft type system for Scheme. In *ACM SIGPLAN Lisp Pointers*, Vol. 7. ACM, 250–262.

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*. https://doi.org/10.1145/2509578.2509581