Julia: An Efficient Dynamic Language for Technical Computing

by

Jeffrey Werner Bezanson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author .....
Department of Electrical Engineering and Computer Science
May 16, 2012

Certified by .....
Alan Edelman
Professor
Thesis Supervisor

Accepted by .....
Leslie Kolodziejski
Chairman, Department Committee on Graduate Students

Julia: An Efficient Dynamic Language for Technical Computing

by

Jeffrey Werner Bezanson

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Dynamic programming languages have become popular for scientific computing. They are generally considered highly productive, but lacking in performance. This thesis presents a new dynamic language for technical computing, designed for performance from the beginning by adapting and extending modern programming language techniques. A design based on generic functions and a rich type system simultaneously enables an expressive programming model and successful type inference, leading to good performance for a wide range of programs. In our system, more behavior can be defined in libraries and user code, allowing our infrastructure to be shared across disciplines.

Thesis Supervisor: Alan Edelman
Title: Professor

Acknowledgments

I thank my advisor, Professor Alan Edelman, for his vision in supporting this project and for his encouragement throughout. His rare combination of theoretical insight and enthusiasm for high-impact software projects has made this work possible.

Julia turned out to be much more than a personal project, owing chiefly to the work of my collaborators Stefan Karpinski and Viral Shah, who wrote most of the standard library and contributed to the language design. I am also personally indebted to them for spurring me to work hard, and for their camaraderie.

Contents

Chapter 1

Introduction

Convenience is winning. Despite continued advances in compiler technology and execution frameworks for high-performance computing, programmers routinely use high-level dynamic languages for algorithm development in applied math and the sciences. These systems (prominent examples include Python [44], R [30], MATLAB®, Octave [39], and SciLab [27]) have greatly increased productivity, but are known to lack performance for many demanding applications. The result is a two-tiered software world, where C and FORTRAN are used for key libraries and production code, while high-level languages are used for interaction and scripting overall workflow. In this thesis I argue that a new approach to dynamic language design can change this situation, providing productivity and performance at once. We should embrace the emerging preference for "scripting" style languages, and ask how these systems can better provide for the future of technical computing.

The "two-tier" architecture, for example writing an application in Python with performance-critical code written in C, seems like a good way to balance performance and productivity, but there are many reasons to move away from it. Naturally, it would be preferable to write compute-intensive code in a more productive language as well, especially when developing parallel algorithms, where code complexity can increase dramatically. Programming in two languages can also be more complex than using either language by itself, due to interfacing issues such as converting between type domains and handling memory reclamation. These interfacing issues

may also add overhead when calling between layers. When such a system is used for mathematical programming there is pressure to write "vectorized" code, which is not natural for every problem. Lastly, from a compiler's point of view, these designs make it difficult to perform whole-program optimization. It is difficult to perform domain-specific optimizations of C code, and expressing algorithms at a higher level makes certain optimizations easier.

Fortunately, there has been significant progress in improving the performance of dynamic languages. Projects like the Python compiler framework PyPy [8] have been fairly successful. Similar efforts exist for languages from LISP onward. The common feature of all such projects is that they seek to add performance to an existing language. This is obviously useful, but we are somewhat surprised to find it has not led to the desired situation outlined above. Julia is designed for performance from the beginning, and we feel this seemingly-subtle difference turns out to be crucial.

"Built-in" performance means that the compiler's type machinery is also available within the language, adding expressiveness. This, in turn, allows more functionality to be implemented in libraries. Many of the key differences between languages used by different disciplines (e.g. R for statistics) could be expressed in libraries, but are instead either part of the language core, or implemented in C where they are more difficult to modify or extend. When optimizers for these languages are developed, knowledge of key library functions often must be encoded into the compiler. Even Common LISP, for which there are several highly-optimizing compilers, specifies arithmetic in the language, and yet users do not all agree on how arithmetic should behave. Some users require specialized types such as fixed-point numbers or intervals, or support for "missing data" values as in R.

Julia has the potential to solve this problem by providing infrastructure that can be shared across domains, without sacrificing the ease and immediacy of current popular systems. We take advantage of, and validate, this infrastructure by writing Julia's standard library in the language itself, which (1) makes the code more generic and increases our productivity, (2) allows inlining library code into user code and vice-versa, and (3) enables direct type analysis of the library instead of requiring

knowledge of library functions to be built in to the compiler. New users are able to read the standard library code, and modify it or imitate it for their own purposes.

Many of the ideas explored here are not exclusively applicable to technical computing, but we have chosen to target that application area for several reasons. First, technical computing has unique concerns that can be especially awkward or inefficient to handle in existing dynamic languages. Examples include the need for a wide variety of numeric types, and the need for efficient arrays of those types. Second, the performance of high-level technical computing languages has begun to seriously lag behind that of more mainstream languages (notably JavaScript), creating a present need for attempts to improve the situation. General-purpose languages like Java or perhaps even JavaScript could be used for technical computing, but we feel the community will continue to prefer environments that cater to its syntactic needs, and are able to prioritize issues of numerical accuracy and performance.

1.1 The Essence of Julia

Julia's primary means of abstraction is dynamic multiple dispatch. Much of a language consists of mechanisms for selecting code to run in different situations — from method selection to instruction selection. We use only dynamic multiple dispatch for this purpose, which is possible through sufficiently expressive dispatch rules. To add usability to this flexibility, types can generally be ignored when not used to specify dispatch behavior.

Types may optionally be used to make declarations, which are considered by the compiler and checked at run time when necessary. However, we do not require declarations for performance. To achieve this, Julia's compiler automatically specializes methods for types encountered at run time (or at compile time, to the extent types are known then). Effectively, every method is a template (in the C++ sense) by default, with parameterization and instantiation directed by the compiler. We feel this design is in line with a general trend towards automation in compiler and language design.

1.2 Contributions of this Thesis

This thesis describes and evaluates the design and implementation of a dynamically-typed language intended to be more amenable to analysis and optimization, and demonstrates its benefits for technical computing applications. Our design allows more behavior to be specified in libraries than existing dynamic languages, and with better performance in many cases. We present key algorithms used in our implementation.

1.3 Organization of this Thesis

Chapter 2 presents the design of Julia: the basic elements of the language, what it looks like from the user's perspective, and the reasoning behind the design. Chapter 3 provides details of the algorithms and implementation that make Julia work effectively. Chapter 4 discusses some features of the standard library, focusing on use cases that highlight Julia's strengths. Chapter 5 presents some points of evaluation. Chapter 6 describes prior research in the area and how Julia relates to it. Chapter 7 concludes, and discusses the status of Julia as an open-source project beginning to attract outside interest.

Chapter 2

Language Design

Static typing appears to have many advantages from an objective, theoretical standpoint: earlier error detection, generally better performance, and support for more accurate tools are often cited in this context. Nevertheless, developers are "voting with their code" for languages that lack strong static typing disciplines. This is the phenomenon that Julia's design addresses, and as such we must present our view of the advantages of dynamic languages. In particular, we do not assume that *every* feature of these languages is equally important. We hypothesize that the following forms of "dynamism" are the most useful:

- The ability to run code at load time and compile time, eliminating some of the distractions of build systems and configuration files.
- A universal `Any` type as the only true static type, allowing the issue of static types to be ignored when desired.
- Never rejecting code that is syntactically well-formed.
- Behavior that depends only on run-time types (unlike, for example, C++, where virtual methods are dispatched by run-time type and function overloads are dispatched by static type).

We explicitly forgo the following features in the interest of preserving the possibility of static typing in a reasonably broad category of situations:

- Types themselves are immutable.
- The type of a value cannot change over its lifetime.
- Local variable environments are not reified.
- Program code is immutable, but new code may be generated and executed at any time.
- Not all bindings are mutable (`const` identifiers are allowed).

This set of restrictions allows the compiler to see all uses of local variables, and perform dataflow analysis on local variables using only local information. This is important, since it allows user code to call statically-unknown functions without interfering with optimizations done around such call sites. Statically-unknown function calls arise in many contexts, such as calling a function taken from an untyped data structure, or dynamically dispatching a method call due to unknown argument types.

The core Julia language contains the following components:

1. A syntax layer, to translate surface syntax to a suitable intermediate representation (IR).
2. A symbolic language and corresponding data structures for representing certain kinds of types, and implementations of lattice operators (*meet*, *join*, and $\leq$) for those types.
3. An implementation of generic functions and dynamic multiple dispatch based on those types.
4. Compiler intrinsic functions for accessing the object model (type definition, method definition, object allocation, element access, testing object identity, and accessing type tags).
5. Compiler intrinsic functions for native arithmetic, bit string operations, and calling native (C or FORTRAN) functions.

6. A mechanism for binding top-level names.

The IR describes a function body as a sequence of assignment operations, function calls, labels, and conditional branches. Julia's semantics are those of a standard imperative language: statements are executed in order, with function arguments evaluated eagerly. All values are conceptually references, and are passed by reference as in LISP.

Julia's core evaluation semantics are particularly bland, because all of the interesting work has been moved to the generic function system. Every function definition is actually a definition of a method for some generic function for some combination of argument types. The "feel" of the language derives mostly from the fact that every function call is dynamically dispatched to the most specific matching method definition, based on the types of all arguments.

2.1 Types

Julia uses dynamic typing, which means that the universal type `Any` is the only static type. Our design philosophy is that types should be quite powerful and expressive, but nearly invisible to the user. Julia programmers must be able to ignore the type system completely if they do not wish to make explicit use of its functionality.

Julia treats types as symbolic descriptions of sets of values. Every value has a unique, immutable, run-time implementation type. Objects carry type tags, and types themselves are Julia objects that can be manipulated at run time. Julia has five kinds of types:

1. abstract types, which may have explicitly-declared subtypes and supertypes
2. composite types (similar to C structs), which have named fields and explicitly-declared supertypes
3. bits types, whose values are represented as bit strings, and which may have explicitly-declared supertypes

4. tuples, immutable ordered collections of values. The type of a tuple is defined recursively as a tuple of the types of the elements. Tuple types are covariant in their element types. A tuple is used to represent the type of a method's arguments.

5. union types, abstract types constructed from other types via set union

Abstract types, composite types, and bits types may have parameters, which makes it possible to express variants of a given type (for example, array types with different element types). These types are all invariant with respect to their parameters (i.e. two versions of the same type with different parameters are simply different, and have no subtype or supertype relationship). Type constructors are applied using curly braces, as in `Array{Float64,1}` (the `Array` type is parameterized by element type and rank).

Bits types allow users to add new fixed-width number-like types and obtain the same performance that primitive numeric types enjoy in other systems. Julia's "built in" numeric types are defined as bits types. Julia method dispatch is based on types rather than field lookup, so whether a value is of a bits type or composite type is a representation detail that is generally invisible.

As an extra complexity, tuple types may end in a special `...` type that indicates any number of elements may be added. This is used to express the types of variadic methods. For example the type `(String, Int...)` indicates a tuple where the first element is a String and any number of trailing integers may be present.

Union types are used primarily to construct tight least upper bounds when the inference algorithm needs to join unrelated types. For example, a method might return an `Int` or a `String` in separate arms of a conditional. In this case its type can be inferred as `Union(Int,String)`. Union types are also useful for defining ad-hoc type hierarchies different from those imagined when the types involved were first defined. For example, we could use `Union(Int,RangeInt)` as the type of array indexes, even though there is no `Index` type inherited by both constituents. Lastly, union types can be used to declare methods applicable to multiple types.

The key to the utility of Julia's type system is its implementation of two important functions: the subtype predicate, which determines whether one type is a subset of another, and type intersection, which computes a type that is a subtype of two given types. These functions form the basis of method dispatch logic and type inference.

2.1.1 Notational Conveniences

An important goal is for users to be able to write Julia programs with virtually no knowledge of type system details. Therefore we allow writing parametric types without parameters, or omitting trailing parameters. `Array` refers to any kind of dense array, `Array{Float64}` refers to a Float64 Array of any rank, and `Array{Float64,2}` refers to a 2-dimensional Float64 Array.

This design also makes it easy to add parameters to types later; existing code does not need to be modified.

2.1.2 Standard Type Hierarchy

Here we present an excerpt from the standard library, showing how a few important types are defined. The fields of composite types are redacted for the sake of brevity. The `<:` syntax indicates a declared subtype relation.

```
abstract Type{T}
type AbstractKind  <: Type; end
type BitsKind      <: Type; end
type CompositeKind <: Type; end
type UnionKind     <: Type; end

abstract Number
abstract Real     <: Number
abstract Float    <: Real
abstract Integer  <: Real
abstract Signed   <: Integer
abstract Unsigned <: Integer

bitstype 32 Float32 <: Float
bitstype 64 Float64 <: Float
```

```
bitstype 8  Bool <: Integer
bitstype 32 Char <: Integer

bitstype 8  Int8   <: Signed
bitstype 8  Uint8  <: Unsigned
bitstype 16 Int16  <: Signed
bitstype 16 Uint16 <: Unsigned
bitstype 32 Int32  <: Signed
bitstype 32 Uint32 <: Unsigned
bitstype 64 Int64  <: Signed
bitstype 64 Uint64 <: Unsigned

abstract AbstractArray{T,N}
type Array{T,N} <: AbstractArray{T,N}; end
```

2.2 Syntax

Julia has a simple block-structured syntax, with notation for type and method definition, control flow, and special syntax for important operators.

2.2.1 Method Definition

Method definitions have a long (multi-line) form and a short form.

```
function iszero(x::Number)
    return x==0
end

iszero(x) = (x==0)
```

A type declaration with `::` on an argument is a dispatch specification. When types are omitted, the default is `Any`. A `::` expression may be added to any program expression, in which case it acts as a run-time type assertion. As a special case, when `::` is applied to a variable name in statement position (a construct which otherwise has no effect) it means the variable *always* has the specified type, and values will be converted to that type (by calling `convert`) on assignment to the variable.

Note that there is no distinct type context; types are computed by ordinary expressions evaluated at run time. For example, `f(x)::Int` is lowered to the function call `typeassert(f(x),Int)`.

Anonymous functions are written using the syntax `x->x+1`.

Local variables are introduced implicitly by assignment. Modifying a global variable requires a `global` declaration.

Operators are simply functions with special calling syntax. Their definitions look the same as those of ordinary functions, for example `+(x,y) = ...`, or `function +(x,y)`.

When the last argument in a method signature is followed by `...` the method accepts any number of arguments, and the last argument name is bound to a tuple containing the tail of the argument list. The syntax `f(t...)` "splices" the contents of an iterable object `t` as the arguments to `f`.

Parametric Methods

It is often useful to refer to parameters of argument types inside methods, and to specify constraints on those parameters for dispatch purposes. Method parameters address these needs. These parameters behave a bit like arguments, but they are always derived automatically from the argument types and not specified explicitly by the caller. The following signature presents a typical example:

```
function assign{T<:Integer}(a::Array{T,1}, i, n::T)
```

This signature is applicable to 1-dimensional arrays whose element type is some kind of integer, any type of second argument, and a third argument that is the same type as the array's element type. Inside the method, `T` will be bound to the array element type.

The primary use of this construct is to write methods applicable to a family of parametric types (e.g. all integer arrays, or all numeric arrays) despite invariance. The other use is writing "diagonal" constraints as in the example above. Such diagonal constraints significantly complicate the type lattice operators.

2.2.2 Control Flow

```
if condition1 || (a && b)
    # single line comment
elseif !condition2
else
    # otherwise
end

while condition
    # body
end

for i in range
    # body
end
```

A `for` loop is translated to a while loop with method calls according to the iteration interface (`start`, `done`, and `next`):

```
state = start(range)
while !done(range, state)
  (i, state) = next(range, state)
  # body
end
```

This design for iteration was chosen because it is not tied to mutable heap-allocated state, such as an iterator object that updates itself.

2.2.3 Special Operators

Special syntax is provided for certain functions.

surface syntax	lowered form
`a[i, j]`	`ref(a, i, j)`
`a[i, j] = x`	`assign(a, x, i, j)`
`[a; b]`	`vcat(a, b)`
`[a, b]`	`vcat(a, b)`
`[a b]`	`hcat(a, b)`
`[a b; c d]`	`hvcat((2,2), a, b, c, d)`

2.2.4 Type Definition

```
# abstract type
abstract Complex{T<:Real} <: Number

# composite type
type ComplexPair{T<:Real} <: Complex{T}
    re::T
    im::T
end

# bits type
bitstype 128 Complex128 <: Complex{Float64}

# type alias
typealias TwoOf{T} (T,T)
```

Constructors

Composite types are applied as functions to construct instances. The default constructor accepts values for each field as arguments. Users may override the default constructor by writing method definitions with the same name as the type inside the `type` block. Inside the `type` block the identifier `new` is bound to a pseudofunction that actually constructs instances from field values. The constructor for the `Rational` type is a good example:

```
type Rational{T<:Integer} <: Real
    num::T
    den::T

    function Rational(num::T, den::T)
        if num == 0 && den == 0
            error("invalid rational: 0//0")
        end
        g = gcd(den, num)
        new(div(num, g), div(den, g))
    end
end
```

This allows `Rational` to enforce representation as a fraction in lowest terms.

2.3 Generic Functions

The vast majority of Julia functions (in both the library and user programs) are generic functions, meaning they contain multiple definitions or methods for various combinations of argument types. When a generic function is applied, the most specific definition that matches the run-time argument types is invoked. Generic functions have appeared in several object systems in the past, notably CLOS [22] and Dylan [41]. Julia is distinguished from these in that it uses generic functions as its primary abstraction mechanism, putting it in the company of research languages like Diesel [13]. Aside from being practical for highly polymorphic mathematical styles of programming, as we will discuss, this design is satisfying also because it permits expression of most of the popular patterns of object-oriented programming, while leaving the core language with fewer distinct features.

Generic functions are a natural fit for mathematical programming. For example, consider implementing exponentiation (the `^` operator in Julia). This function lends itself to multiple definitions, specializing on both arguments separately: there might be one definition for two floating-point numbers that calls a standard math library routine, one definition for the case where the second argument is an integer, and separate definitions for the case where the first argument is a matrix. In Julia these signatures would be written as follows:

```
function ^(x::Float64, p::Float64)
function ^(x, p::Int)
function ^(x::Matrix, p)
```

2.3.1 Singleton Kinds

A generic function's method table is effectively a dictionary where the keys are types. This suggests that it should be just as easy to define or look up methods with types themselves as with the types of values. Defining methods on types directly is analogous to defining class methods in class-based object systems. With multi-methods,

definitions can be associated with combinations of types, making it easy to represent properties not naturally owned by one type.

To accomplish this, we introduce a special singleton kind `Type{T}`, which contains the type `T` as its only value. The result is a feature similar to `eql` specializers in CLOS, except only for types. An example use is defining type traits:

```
typemax(::Type{Int64}) = 9223372036854775807
```

This definition will be invoked by the call `typemax(Int64)`. Note that the name of a method argument can be omitted if it is not referenced.

Types are useful as method arguments in several other cases. One example is file I/O, where a type can be used to specify what to read. The call `read(file,Int32)` reads a 4-byte integer and returns it as an `Int32` (a fact that the type inference process is able to discover). We find this more elegant and convenient than systems where enums or special constants must be used for this purpose, or where the type information is implicit (e.g. through return-type overloading).

2.3.2 Method Sorting and Ambiguity

Methods are stored sorted by specificity, so the first matching method (as determined by the subtype predicate) is always the correct one to invoke. This means much of the dispatch logic is contained in the sorting process. Comparing method signatures for specificity is not trivial. As one might expect, the "more specific"[1] predicate is quite similar to the subtype predicate, since a type that is a subtype of another is indeed more specific than it. However, a few additional rules are necessary to capture the intuitive concept of "more specific". In fact until this point "more specific" has had no formal meaning; its formal definition is summarized as the disjunction of the following rules (A is more specific than B if):

[1] Actually, "not less specific", since specificity is a partial order.

1. A is a subtype of B

2. A is of the form `T{P}` and B is of the form `S{Q}`, and T is a subtype of S for some parameter values

3. The intersection of A and B is nonempty, more specific than B, and not equal to B, and B is not more specific than A

4. A and B are tuple types, A ends in a vararg (`...`) type, and A would be more specific than B if its vararg type were expanded to give it the same number of elements as B

Rule 2 means that declared subtypes are always more specific than their declared supertypes regardless of type parameters. Rule 3 is mostly useful for union types: if A is `Union(Int32,String)` and B is `Number`, A should be more specific than B because their intersection (`Int32`) is clearly more specific than B. Rule 4 means that argument types are more important for specificity than argument count; if A is `(Int32...)` and B is `(Number, Number)` then A is more specific.

Julia uses *symmetric* multiple dispatch, which means all argument types are equally important. Therefore, ambiguous signatures are possible. For example, given `foo(x::Number, y::Int)` and `foo(x::Int, y::Number)` it is not clear which method to call when both arguments are integers. We detect ambiguities when a method is added, by looking for a pair of signatures with a non-empty intersection where neither one is more specific than the other. A warning message is displayed for each ambiguity, showing the user the computed type intersection so it is clear what definition is missing. For example:

```
Warning: New definition foo(Int,Number) is ambiguous with foo(Number,Int).
        Make sure foo(Int,Int) is defined first.
```

2.4 Intrinsic Functions

The run-time system contains a small number of primitive functions for tasks like determining the type of a value, accessing fields of composite types, and constructing values of each of the supported kinds of concrete types. There are also arithmetic intrinsics corresponding to machine-level operations like fixed-width integer addition, bit shifts, etc. These intrinsic functions are implemented only in the code generator and do not have callable entry points. They operate on bit strings, which are not first class values but can be converted to and from Julia bits types via boxing and unboxing operations.

In our implementation, the core system also provides functions for constructing arrays and accessing array elements. Although this is not strictly necessary, we did not want to expose unsafe memory operations (e.g. load and store primitives) in the language. In an earlier implementation, the core system provided a bounds-checked `Buffer` abstraction, but having both this type and the user-level `Array` type proved inconvenient and confusing.

2.5 Design Limitations

In our design, type information always flows along with values, in the forward control flow direction. This prevents us from doing certain tricks that static type systems are capable of, such as return-type overloading. Return-type overloading requires a robust notion of the type of a value *context*—the type expected or required of some term—in order to select code on that basis. There are other cases where "backwards" type flow might be desirable, such as determining the type of a container based on the type of a value stored into it at a later program point. It may be possible to get around this limitation in the future using inversion of control—passing a function argument whose result type has already been inferred, and using that type to construct a container before elements are computed.

Modularity is a perennial difficulty with multiple dispatch, as any function may

apply to any type, and there is no point where functions or types are closed to future definitions. Thus at the moment Julia is essentially a whole-program compiler. We plan to implement a module system that will at least allow code to control which name bindings and definitions it sees. Such modules could be separately compiled to the extent that programmers are willing to ask for their definitions to be "closed".

Lastly, at this time Julia uses a bit more memory than we would prefer. Our compiler data structures, type information, and generated native code take up more space than the compact bytecode representations used by many dynamic languages.

Chapter 3

Implementation

3.1 Method Dispatch

Much of the implementation is organized around method dispatch. The dispatch logic is both a large portion of the behavior of Julia functions, and the entry point of the compiler's type inference and specialization logic.

3.1.1 Method Caching and Specialization

The first step of method dispatch is to look for the argument types in a per-function cache. The cache has an entry for (almost) every set of concrete types to which the function has been applied. Concrete types are hash-consed, so they can be compared by simple pointer comparison. This makes cache lookup faster than the subtype predicate. As part of hash-consing, concrete types are assigned small integer IDs. The ID of the first argument is used as a primary key into a method cache, so when signatures differ only in the type of the first argument a simple indexed lookup suffices.

On a cache miss, a slower search for the matching definition is performed using subtype. Then, type inference is invoked on the matching method using the types of the actual arguments. The resulting type-annotated and optimized method is stored in the cache. In this way, method dispatch is the primary source of type information for the compiler.

3.1.2 Method Specialization Heuristics

Our aggressive use of code specialization has the obvious pitfall that it might lead to excessive code generation, consuming memory and compile time. We found that a few mild heuristics suffice to give a usable system with reasonable resource requirements.

The first order of business is to ensure that the dispatch and specialization process converges. The reason it might not is that our type inference algorithm is implemented in Julia itself. Calling a method on a certain type A can cause the type inference code to call the same method on type B, where types A and B follow an infinite ascending chain in either of two partial orders (the typeof order or the subtype order). Singleton kinds are the most prominent example, as type inference might attempt to successively consider `Int32`, `Type{Int32}`, `Type{Type{Int32}}`, and so on. We stop this process by replacing any nestings of `Type` with the unspecialized version of `Type` during method specialization (unless the original method declaration actually specified a type like `Type{Type{Int32}}`).

The next heuristic avoids specializing methods for tuple types of every length. Tuple types are cached as the intersection of the declared type of the method slot with the generic tuple type `(Any...)`. This makes the resulting cache entry valid for any tuple argument, again unless the method declaration contained a more specific tuple type. Note that all of these heuristics require corresponding changes in the method cache lookup procedure, since they yield cache entries that do not have to exactly match candidate arguments.

A similar heuristic is applied to variadic methods, where we wish to avoid caching argument lists of every length. This is done by capping argument lists at the length of the longest signature of any method in the same generic function. The "capping" involves replacing the last argument with a `...` type. Ideally, we want to form the biggest type that's not a supertype of any other method signatures. However, this is not always possible and the capped type might conflict with another signature. To deal with this case, we find all non-empty intersections of the capped type with other signatures, and add dummy cache entries for them. Hitting one of these entries

alerts the system that the arguments under consideration are not really in the cache. Without the dummy entries, some arguments might incorrectly match the capped type, causing the wrong method to be invoked.

The next heuristic concerns singleton kinds again. Because of the singleton kind feature, every distinct type object (`Any`, `Number`, `Int`, etc.) passed to a method might trigger a new specialization. However, most methods are not "class methods" and are not concerned with type objects. Therefore, if no method definition in a certain function involves `Type` for a certain argument slot, then that slot is not specialized for different type objects.

Finally, we introduce a special type `ANY` that can be used in a method signature to hint that a slot should not be specialized. This is used in the standard library in a small handful of places, and in practice is less important than the heuristics described above.

3.2 Type Inference

Types of program expressions and variables are inferred by forward dataflow analysis[1]. A key feature of this form of type inference is that variable types are inferred at each use, since assignment is allowed to change the type of a variable. We determine a maximum fixed-point (MFP) solution using Algorithm 1, based on Mohnen's graph-free dataflow analysis framework [38]. The basic idea is to keep track of the state (the types of all variables) at each program point, determine the effect of each statement on the state, and ensure that type information from each statement eventually propagates to all other statements reachable by control flow. We augment the basic algorithm with support for mutually-recursive functions (functions are treated as program points that might need to be revisited).

The origin of the type information used by the MFP algorithm is evaluation of known functions over the type domain [18]. This is done by the eval subroutine. The

[1]Adding a reverse dataflow pass could potentially improve type information, but we have not yet done this.

interpret subroutine calls eval, and also handles assignment statements by returning the new types of affected variables. Each known function call is either to one of the small number of built-in functions, in which case the result type is computed by a (usually trivial) hand-written type transfer function, or to a generic function, in which case the result type is computed by recursively invoking type inference. In the generic function case, the inferred argument types are met ($\sqcap$) with the signatures of each method definition. Matching methods are those where the meet (greatest lower bound) is not equal to the bottom type (`None` in Julia). Type inference is invoked on each matching method, and the results are joined ($\sqcup$) together. The following equation summarizes this process:

$$T(f, t_{arg}) = \bigsqcup_{(s,g) \in f} T(g, t_{arg} \sqcap s)$$

T is the type inference function. t_{arg} is the inferred argument tuple type. The tuples (s, g) represent the signatures s and their associated definitions g within generic function f.

Two optimizations are helpful here. First, it is rarely necessary to consider all method definitions. Since methods are stored in sorted order, as soon as the union of the signatures considered so far is a supertype of t_{arg}, no more definitions need to be considered. Second, the join operator employs *widening* [19]: if a type becomes too large it may simply return `Any`. In this case the recursive inference process may stop immediately.

3.2.1 Interprocedural Type Inference

Type inference is invoked through "driver" Algorithm 2 which manages mutual recursion and memoization of inference results. A stack of abstract activation records is maintained and used to detect recursion. Each function has a property incomplete(F, A) indicating that it needs to be revisited when new information is discovered about the result types of functions it calls. The incomplete flags collectively represent a set analogous to W in Algorithm 1.

Algorithm 1 Infer function return type

Input: function F, argument type tuple A, abstract execution stack T
Output: result type $T.R$

$V \leftarrow$ set of all locally-bound names
$V_a \leftarrow$ argument names
$n \leftarrow \text{length}(F)$
$W \leftarrow \{1\}$ {set of program counters}
$P_r \leftarrow \emptyset$ {statements that recur}
$\forall i, S[1, V[i]] \leftarrow$ Undef
$\forall i, S[1, V_a[i]] \leftarrow A[i]$
while $W \neq \emptyset$ **do**
  $p \leftarrow \text{choose}(W)$
  repeat
    $W \leftarrow W - p$
    $new \leftarrow \text{interpret}(F[p], S[p], T)$
    if $T.rec$ **then**
      $P_r \leftarrow P_r \cup \{p\}$
      $T.rec \leftarrow$ false
    end if
    $p\prime \leftarrow p + 1$
    if $F[p] =$`(goto l)` **then**
      $p\prime \leftarrow l$
    else if $F[p] =$`(gotoif cond l)` **then**
      if not $new \leq S[l]$ **then**
        $W \leftarrow W \cup \{l\}$
        $S[l] \leftarrow S[l] \sqcup new$
      end if
    else if $F[p] =$`(return e)` **then**
      $p\prime \leftarrow n + 1$
      $r \leftarrow \text{eval}(e, S[p], T)$
      if not $r \leq T.R$ **then**
        $T.R \leftarrow T.R \sqcup r$
        $W \leftarrow W \cup P_r$
      end if
    end if
    if $p\prime \leq n$ **and not** $new \leq S[p\prime]$ **then**
      $S[p\prime] \leftarrow S[p\prime] \sqcup new$
      $p \leftarrow p\prime$
    end if
  until $p\prime = n + 1$
end while
$T.rec \leftarrow P_r \neq \emptyset$

The outer loop in Algorithm 2 looks for an existing activation record for its input function and argument types. If one is found, it marks all records from that point to the top of the stack, identifying all functions involved in the call cycle. These marks are discovered in Algorithm 1 when interpret returns, and all affected functions are considered incomplete. Algorithm 2 continues to re-run inference on incomplete functions, updating the inferred result type, until no recursion occurs or the result type converges.

Algorithm 2 Interprocedural type inference

Input: function F, argument type tuple A, abstract execution stack S
Output: returned result type

```
R ← ⊥
if recall(F, A) exists then
  R ← recall(F, A)
  if not incomplete(F, A) then
    return R
  end if
end if
f ← S
while not empty(f) do
  if f.F is F and f.A = A then
    r ← S
    while not r = tail(f) do
      r.rec ← true
      r ← tail(r)
    end while
    return f.R
  end if
  f ← tail(f)
end while
T ← extend(S, Frame(F = F, A = A, R = R, rec = false))
invoke Algorithm 1 on F, A, T
recall(F, A) ← T.R
incomplete(F, A) ← (T.rec ∧ ¬(R = T.R))
return T.R
```

Because this algorithm approximates run-time behavior, we are free to change it without affecting the behavior of user programs — except that they might run faster. One valuable improvement is to attempt full evaluation of branch conditions, and remove branches with constant conditions.

3.3 Lattice Operations

Our type lattice is complicated by the presence of type parameters, unions, and diagonal type constraints in method signatures. Fortunately, for our purposes only the $\leq$ (subtype) relation needs to be computed accurately, as it bears final responsibility for whether a method is applicable to given arguments. Type union and intersection, used to estimate least upper bounds and greatest lower bounds, respectively, may both be conservatively approximated. If their results are too coarse, the worst that can happen is performing method dispatch or type checks at run time, since the inference process will simply conclude that it does not know precise types.

A complication arises from the fact that our abstract domain is available in a first-class fashion to user programs. When a program contains a type-valued expression, we want to know which type it will evaluate to, but this is not possible in general. Therefore in addition to the usual *type imprecision* (not knowing the type of a value), we must also model *type uncertainty*, where a type itself is known imprecisely. A common example is application of the `typeof` primitive to a value of imprecise type. What is the abstract result of `typeof(x::Number)`? We handle this with a special type kind that represents a *range* rather than a point within the type lattice. These kinds are essentially the type variables used in bounded polymorphism [10]. In this example, the transfer function for `typeof` is allowed to return `Type{T<:Number}`, where `T` is a new type variable.

3.3.1 Subtype Predicate

See Algorithm 3. Note that extensional type equality can be computed as $(A \leq B \wedge B \leq A)$, and this is used for types in invariant context (i.e. type parameters). The algorithm uses subroutines $\mathrm{p}(A)$ which gives the parameters of type A, and $\mathrm{super}(A)$ which gives the declared supertype of A.

Algorithm 3 Subtype

Input: types A and B
Output: $A \leq B$

```
if A is a tuple type then
  if B is not a tuple type then
    return false
  end if
  for i = 1 to length(A) do
    if A[i] is T... then
      if last(B) exists and is not S... then
        return false
      end if
      return subtype(T, B[j])), i ≤ j ≤ length(B)
    else if i > length(B) or not subtype(A[i], B[i]) then
      return false
    else if B[i] is T... then
      return subtype(A[j], T)), i < j ≤ length(A)
    end if
  end for
else if A is a union type then
  return ∀t ∈ A, subtype(t, B)
else if B is a union type then
  return ∃t ∈ B, subtype(A, t)
end if
while A ≠ Any do
  if typename(A) = typename(B) then
    return subtype(p(A), p(B)) ∧ subtype(p(B), p(A))
  end if
  A ← super(A)
end while
if A is of the form Type{T} then
  return subtype(typeof(p(A)[1]), B)
else if B is of the form Type{T} then
  B ← p(B)[1]
  return subtype(A, B) ∧ subtype(B, A)
end if
return B = Any
```

3.3.2 Type Union

Since our type system explicitly supports unions, the union of T and S can be computed simply by constructing the type `Union(T,S)`. An obvious simplification is performed: if one of T or S is a subtype of the other, it can be removed from the union. Nested union types are flattened, followed by pairwise simplification.

3.3.3 Type Intersection

This is the difficult one: given types T and S, we must try to compute the smallest type R such that $\forall s, s \in T \wedge s \in S \Rightarrow s \in R$. The conservative solution is to give up on finding the smallest such type, and return *some* type with this property. Simply returning T or S suffices for correctness, but in practice this algorithm makes the type inference process nearly useless. A slightly better algorithm is to check whether one argument is a subtype of the other, and return the smaller type. It is also possible to determine quickly, in many cases, that two types are disjoint, and return $\perp$. With these two enhancements we start to obtain some useful type information. However, we need to do much better to take full advantage of the framework set up so far.

Our algorithm has two phases. First, the structures of the two input types are analyzed in a manner similar to subtype, except a constraint environment is built, with entries $T \leq S$ for type variables T in covariant contexts (tuples) and entries $T = S$ for type variables T in invariant contexts (type parameters). In the second phase the constraints are solved with an algorithm (Algorithm 5) similar to that used by traditional polymorphic type systems [40].

The code for handling tuples and union types is similar to that in Algorithm 3, so we focus instead on intersecting types in the nominative hierarchy (Algorithm 4). The base case occurs when the input types are from the same family, i.e. have the same typename. All we need to do is visit each parameter to collect any needed constraints, and otherwise check that the parameters are equal. When a parameter is a type variable, it is effectively covariant, and must be intersected with the corresponding parameter of the other type to form the final result.

Algorithm 4 Intersection of nominative types

Input: types A and B, current constraint environment
Output: return T such that $A \sqcap B \leq T$, updated environment

```
if typename(A) = typename(B) then
  pa ← copy(p(A))
  for i = 1 to length(p(A)) do
    if p(A)[i] is a typevar then
      add (p(A)[i] = p(B)[i]) to constraints
    else if p(B)[i] is a typevar then
      add (p(B)[i] = p(A)[i]) to constraints
    end if
    pa[i] ← intersect(p(A)[i], p(B)[i])
  end for
  return typename(A){pa...}
else
  sup ← intersect(super(A), B)
  if sup = ⊥ then
    sup ← intersect(A, super(B))
    if sup = ⊥ then
      return ⊥
    else
      sub ← B
    end if
  else
    sub ← A
  end if
  E ← conform(sup, super_decl(sub))
  if E contains parameters not in formals(sub) then
    return ⊥
  end if
  return intersect(sub, typename(sub){E...})
end if
```

Algorithm 5 Solve type variable constraints

Input: environment X of pairs $T \leq S$ and $T = S$
Output: environment Y of unique variable assignments $T = S$, or failure

```
Y ← ∅
replace (T ≤ S) ∈ X with (T = S) when S is concrete
for all (T = S) ∈ X do
  if (T = R) ∈ X and S ≠ R then
    return failure
  end if
end for
for all (T ≤ S) ∈ X do
  if (T = U) ∈ X then
    if not find(X, U) ≤ S then
      return failure
    else
      X ← X − (T ≤ S)
    end if
  else if (T ≤ U) ∈ X and U not a variable then
    replace U with U ⊓* S
    X ← X − (T ≤ S)
  end if
end for
for all variables T do
  if (T = U) ∈ X then
    Y ← Y ∪ {T = U}
  else
    S ← ⨅* find(X, U), ∀(T ≤ U) ∈ X
    if S = ⊥ then
      return failure
    end if
    Y ← Y ∪ {T = S}
  end if
end for
```

When the argument types are not from the same family, we recur up the type hierarchy to see if any supertype of one of the arguments matches the other. If so, the recursion gives us the intersected supertype *sup*, and we face the problem of mapping it to the family of the original argument type. To do this, we first call subroutine conform, which takes two types with the same structure and returns an environment E mapping any type variables in one to their corresponding components in the other. super_decl(t) returns the type template used by t to instantiate its supertype. If all goes well, this tells us what parameters *sub* would have to be instantiated with to have supertype *sup*. If, however, E contains type variables not controlled by *sub*, then there is no way a type like *sub* could have the required supertype, and the overall answer is $\bot$. Finally, we apply the base case to intersect *sub* with the type obtained by instantiating its family with parameter values in E.

Constraints $T \leq S$ where S is a concrete type are converted to $T = S$ to help sharpen the result type. If Algorithm 5 identifies any conflicting constraints, the type intersection is empty. If each type variable has exactly one constraint $T = U$, we can simply substitute find(X, U) for each occurrence of T in the computed type intersection, and we have a final answer. find works in the *union-find* sense, following chains of equalities until we hit a non-variable or an unconstrained variable. Unconstrained type variables may be left in place.

The remaining case is type variables with multiple constraints. Finding a satisfying assignment requires intersecting all the upper bounds for a variable. It is here that we choose to throw in the towel and switch to a coarser notion of intersection, denoted by $\sqcap^*$. Intersection is effectively the inner loop of type inference, so in the interest of getting a reasonable answer quickly we might pick $X \sqcap^* Y = X$. A few simple heuristics might as well be added; for example cases like two non-parameterized types where one is an immediate subtype of the other can be supported easily.

In our implementation, type intersection handles most of the complexity surrounding type variables and parametric methods. It is used to test applicability of parametric methods; since all run-time argument lists are of concrete type, intersecting their types with method signatures behaves like subtype, except static parameters

are also properly matched. If intersection returns $\perp$ or does not find values for all static parameters for a method, the method is not applicable. Therefore in practice we do not really have the freedom to implement $\sqcap$ and $\sqcap^*$ any way that obeys our correctness property. They must be at least as accurate as subtype in the case where one argument is concrete.

3.3.4 Widening Operators

Lattices used in practical program analyses often fail to obey the finite chain condition necessary for the MFP algorithm to converge (i.e. they are not of finite height) and ours is no exception.

Widening is applied in two places: by the join operator, and on every recursive invocation of type inference. When a union type becomes too large (as determined by an arbitrarily-chosen cutoff), it is replaced with `Any`. Tuple types lend themselves to two infinite chains: one in depth (`(Any,)`, `((Any,),)`, `(((Any,),),)`, etc.) and one in length (`(Any...,)`, `(Any,Any...,)`, `(Any,Any,Any...,)`, etc.). These chains are capped at arbitrary cutoffs each time the inference process needs to construct a tuple type.

3.4 Code Generation and Optimization

After type inference is complete, we annotate each expression with its inferred type. We then run two symbolic optimization passes. If the inferred argument types in a method call indicate that a single method matches, we are free to inline that method. For methods that return multiple values, inlining often yields expressions that construct tuples and immediately take them apart. The next optimization pass identifies these cases and removes the tuple allocations.

The next set of optimizations is applied during code generation. Our code generator targets the LLVM compiler framework [34]. First, we examine uses of variables and assign local variables specific scalar types where possible (LLVM uses a typed code representation). The `box` operations used to tag bit strings with types are done

lazily; they add a compile-time tag that causes generation of the appropriate allocation code only when the value in question hits a context that requires it (for example, assignment to an untyped data structure, or being passed to an unknown function).

The code generator recognizes calls to key built-in and intrinsic functions, and replaces them with efficient in-line code where possible. For example, the `is` function yields a pointer comparison, and `typeof` might yield a constant pointer value if the type of its argument is known. Calls known to match single methods generate code to call the correct method directly, skipping the dispatch process.

Finally, we run several of the optimization passes provided by LLVM. This gives us all of the standard scalar optimizations, such as strength reduction, dead code elimination, jump threading, and constant folding. When we are able to generate well-typed but messy code, LLVM gets us the rest of the way to competitive performance. We have found that care is needed in benchmarking: if the value computed by a loop is not used, in some cases LLVM has been able to delete the whole thing.

Chapter 4

Example Use Cases

4.1 Numeric Type Promotion

Numeric types and arithmetic are fundamental to all programming, but deserve extra attention in the case of scientific computing. In traditional compiled languages such as C, the arithmetic operators are the most polymorphic "functions", and hence cannot be written in the language itself. Arithmetic must be defined in the compiler, including contentious decisions such as how to handle operations with mixed argument types.

In Julia, multiple dispatch is used to define arithmetic and type promotion behaviors at the library level rather than in the compiler. As a result, the system smoothly incorporates new operators and numeric types with minimal work.

Four key utility functions comprise the type promotion system. For simplicity, we consider only two-argument forms of promotion although multi-argument promotion is also defined and used.

1. `convert(T, value)` converts its second argument to type `T`
2. `promote_rule(T1,T2)` defines which of two types is greater in the promotion partial order
3. `promote_type(T1,T2)` uses `promote_rule` to determine which type should be used for values of types `T1` and `T2`

4. `promote(v1, v2)` converts its arguments to an appropriate type and returns the results

`promote` is implemented as follows:

```
function promote{T,S}(x::T, y::S)
    (convert(promote_type(T,S),x), convert(promote_type(T,S),y))
end
```

`promote_type` simply tries `promote_rule` with its arguments in both orders, to avoid the need for repeated definitions:

```
function promote_type{T,S}(::Type{T}, ::Type{S})
    if applicable(promote_rule, T, S)
        return promote_rule(T,S)
    elseif applicable(promote_rule, S, T)
        return promote_rule(S,T)
    else
        error("no promotion exists for ",T," and ",S)
    end
end
```

`convert` and `promote_rule` are implemented for each type. Two such definitions for the `Complex128` type are:

```
promote_rule(::Type{Complex128}, ::Type{Float64}) = Complex128
convert(::Type{Complex128}, x::Real) = complex128(x, 0)
```

With these definitions in place, a function may gain generic promotion behavior by adding the following kind of definition:

```
+(x::Number, y::Number) = +(promote(x,y)...)
```

This means that, given two numeric arguments where no more specific definition matches, promote the arguments and retry the operation (the `...` "splices" the two values returned by `promote` into the argument list). The standard library contains such definitions for all basic arithmetic operators. For this recursion to terminate, we require only that each `Number` type implement `+` for two arguments of that type, e.g.

```
+(x::Int64, y::Int64) = ...
+(x::Float64, y::Float64) = ...
+(x::Complex128, y::Complex128) = ...
```

Therefore, each new type requires only one definition of each operator, and a handful of `convert` and `promote_rule` definitions. If n is the number of types and m is the number of operators, a new type requires $O(n + m)$ rather than $O(n \cdot m)$ definitions.

The reader will notice that uses of this mechanism involve multiple method calls, as well as potentially expensive features such as tuple allocation and argument splicing. Without a sufficient optimizing compiler, this implementation would be completely impractical. Fortunately, through type analysis, inlining, elision of unnecessary tuples, and lowering of the `apply` operation implied by `...`, Julia's compiler is able to eliminate all of the overhead in most cases, ultimately yielding a sequence of machine instructions comparable to that emitted by a traditional compiler.

The most troublesome function is `promote_type`. For good performance, we must elide calls to it, but doing so may be incorrect since the function might throw an error. By fortunate coincidence though, the logic in `promote_type` exactly mirrors the analysis done by type inference: it only throws an error if no matching methods exist for its calls to `promote_rule`, in which case type inference concludes that the function throws an error regardless of which branch is taken. `applicable` is a built-in function known to be free of effects. Therefore, whenever a sharp result type for `promote_type` can be inferred, it is also valid to remove the unused arms of the conditional.

4.2 Code Generation and Staged Functions

The presence of types and an inference pass creates a new, intermediate translation stage which may be customized (macros essentially customize syntax processing, and object systems customize run time behavior). This is the stage at which types are known, and it exists in Julia via the compiler's method specialization machinery. Spe-

cialization may occur at run time during dispatch, or at compile time when inference is able to determine argument types accurately. Running custom code at this stage has two tremendous effects: first, optimized code can be generated for special cases, and second, the type inference system can effectively be extended to be able to make new type deductions relevant to the user's application.

For example, we might want to write functions that apply to two arrays of different dimensionality, where the result has the higher of the two argument dimensionalities. One such function is a "broadcasting" binary elementwise operator, that performs computations such as adding a column vector to every column of a matrix, or adding a plane to every slice of a 3-dimensional dataset. We can determine the shape of the result array with the following function:

```
function promote_shape(s1::Tuple, s2::Tuple)
    if length(s1) > length(s2)
        return s1
    else
        return s2
    end
end
```

The type system can easily express the types of array shapes, for example `(Int,Int)` and `(Int,Int,Int)`. However, inferring a sharp result type for this simple function is still challenging. The inference algorithm would have to possess a theory of the `length` and `>` functions, which is not easily done given that all Julia functions may be redefined and overloaded with arbitrary methods.

One solution might be to allow the user to write some kind of compiler extension or declaration. This approach is not ideal, since it might result in duplicated information, or require the user to know more than they want to about the type system.

Instead, this function can be written as a *staged function* (or more accurately in our case, a *staged method*). This is a function that runs at an earlier translation "stage", i.e. compile time, and instead of returning a result value returns code that will compute the result value when executed [31]. Here is the staged version of

`promote_shape`[1]:

```
@staged function promote_shape(s1::Tuple, s2::Tuple)
    if length(s1) > length(s2)
        quote return s1 end
    else
        quote return s2 end
    end
end
```

The signature of this definition behaves exactly like any other method signature: the type annotations denote run-time types for which the definition is applicable. However, the body of the method will be invoked on the *types* of the arguments rather than actual arguments, and the result of the body will be used to generate a new, more specialized definition. For example, given arguments of types `(Int,Int)` and `(Int,Int,Int)` the generated definition would be:

```
function promote_shape(s1::(Int,Int), s2::(Int,Int,Int))
    return s2
end
```

Observe that the type of this function is trivial to infer.

The staged function body runs as normal user code, so whatever definition of `>` is visible will be used, and the compiler does not have to know how it behaves. Critically, the staged version of the function looks similar to the normal version, requiring only the insertion of `quote` to mark expressions deferred to the next stage.

In the case where a program is already statically-typeable, staged functions preserve that property. The types of the arguments to the staged function will be known at compile time, so the custom code generator can be invoked at compile time. Then the compiler may inline the result or emit a direct call to the generated code, as usual.

Or, if the user does not require static compilation, the custom code generator can be invoked at run time. Its results are cached for each new combination of argument types, so compilation pauses are infrequent.

[1]The `@` denotes a macro invocation. At present, staged methods are implemented by a macro, but full integration into the language is planned.

Thus we have a language with the convenience of run-time-only semantics, which can be compiled just-in-time or ahead-of-time[2] with minimal performance differences, including custom code generation without the need for run-time `eval`. Most importantly, functions with complex type behavior can be implemented in libraries without losing performance. Of course, ordinary Julia functions may also have complex type behavior, and it is up to the library designer to decide which functions should be staged.

4.3 Generic Programming

Support for generic programming is one of Julia's strengths. Code in dynamic languages is often thought of as generic by default, due to the absence of type restrictions, but this has its limits. First, many systems, such as Common LISP, support optional type declarations to improve performance. However, when this feature is used code usually becomes monomorphic as a result. Second, some cases of generic programming require the ability to specify behaviors that *vary* based on types, for example initializing a variable with the right kind of container, or with an appropriate value for different numeric types.

Julia has neither of these problems. The first is solved both by automatic specialization (which usually eliminates the need for performance-seeking declarations), and static parameters, which allow declarations containing type variables. The second is solved by the ability to define type traits. Multiple dispatch is also helpful, as it provides many ways to extend functions in the future.

As an example, here is how `max` can be written for any array:

```
function max{T<:Real}(A::AbstractArray{T})
    v = typemin(T)
    for x in A
        if x > v
            v = x
        end
```

[2]The ahead-of-time compiler is not yet implemented.

```
    end
    return v
end
```

At the same time, we could provide an alternate implementation that makes even fewer assumptions:

```
function max(A)
    v = typemin(eltype(A))
    ...
end
```

This allows any container to expose its element type by implementing `eltype`. Here we do not have to know how the element type is determined. In fact, this version would work for arrays equally well as the first implementation, but we skip the opportunity to specify that it is only defined for arrays with `Real` elements. We might also choose to call `typemin` only if the container is empty, and otherwise initialize `v` with the first element. Then `max` would work on containers that do not implement `eltype`, as long as they are never empty.

Flexible ad-hoc polymorphism plays a significant role in Julia's overall performance. In scientific computing especially, important special cases exist among the myriad datatypes that might appear in a program. Our dispatch model permits writing definitions for more specialized cases than most object-oriented languages.

For example, we have `Array`, the type of dense arrays, and `SubArray`, an abstract array that references a contiguous section of another array. The BLAS and LAPACK libraries allow the caller to specify dimension strides, and we would like to call them for any arrays they can handle. To do this, we can define the following types:

```
typealias Matrix{T}                   Array{T,2}
typealias StridedMatrix{T,A<:Array}   Union(Matrix{T}, SubArray{T,2,A})
```

Then we can write definitions such as `*(StridedMatrix{T},StridedMatrix{T})` for appropriate element types `T`. This replaces the custom dispatch schemes often implemented in array programming systems.

Chapter 5

Evaluation

5.1 Performance

We have compared the execution speed of Julia code to that of six other languages: C++, Python, MATLAB®, Octave, R, and JavaScript. Figure 5-1 [1] shows timings for five scalar microbenchmarks, and two simple array benchmarks. All numbers are ratios relative to the time taken by C++. The first five tests do not reflect typical application performance in each environment; their only purpose is to compare the code executed for basic language constructs manipulating scalar quantities and referencing individual array elements.

We can see why the standard libraries of these environments are developed in C and FORTRAN. MATLAB®has a JIT compiler that works quite well in some cases, but is inconsistent, and performs especially poorly on user-level function calls. The V8 JavaScript JIT compiler's performance is impressive. Anomalously, both Julia and JavaScript seem to beat C++ on pi_sum, but we have not yet discovered why this might be.

[1] These measurements were done by Stefan Karpinski on a MacBook Pro with a 2.53GHz Intel Core 2 Duo CPU and 8GB of 1066MHz DDR3 RAM.

Python 2.7.1, MATLAB®R2011a, Octave 3.4, R 2.14.2, V8 3.6.6.11 C++ compiled by GCC 4.2.1, taking best timing from all optimization levels (-O0 through -O3).

The Python implementations of rand_mat_stat and rand_mat_mul use NumPy (v1.5.1) functions; the rest are pure Python implementations.

Figure 5-1: Microbenchmark results (times relative to C++)

test	Julia	Python	MATLAB®	Octave	R	JavaScript
fib	1.97	31.47	1336.37	2383.80	225.23	1.55
parse_int	1.44	16.50	815.19	6454.50	337.52	2.17
quicksort	1.49	55.84	132.71	3127.50	713.77	4.11
mandel	5.55	31.15	65.44	824.68	156.68	5.67
pi_sum	0.74	18.03	1.08	328.33	164.69	0.75
rand_mat_stat	3.37	39.34	11.64	54.54	22.07	8.12
rand_mat_mul	1.00	1.18	0.70	1.65	8.64	41.79

The rand_mat_stat code manipulates many 5-by-5 matrices. Here the performance gaps close, but the arrays are not large enough for library time to dominate, so Julia's ability to specialize call sites wins the day (despite the fact that most of the array library functions involved are written in Julia itself).

The rand_mat_mul code demonstrates a case where time spent in BLAS [35] dominates. MATLAB®gets its edge from using a multi-threaded BLAS (threading is available in the BLAS Julia uses, but it was disabled when these numbers were taken). R may not be using a well-tuned BLAS in this install; more efficient configurations are probably possible. JavaScript as typically deployed is not able to call the native BLAS code, but the V8 compiler's work is respectable here.

Figure 5-2 [2] compares Julia and Python on some more realistic "task level" benchmarks. The first test defines two data types (classes in Python), then forms (by appending) a heterogeneous array containing one million instances of each type. Then a method is called on each object in the array. On the first run, Julia incurs some compilation overhead and is about 8x faster. On future runs it is up to 16x faster. Although the method calls cannot be optimized, Julia is still able to gain an advantage likely due to use of native arithmetic for looping.

The second test reads each line of a 100000-line, 7MB CSV file, and identifies and separates the comma-delimited fields. Python uses mature C libraries for these

[2]Linux kernel 3.2.8 PC with 3.2GHz Intel Core i5 CPU, 4GB of RAM. Python 2.7.2.

Figure 5-2: Task-level benchmark results (times in seconds)

test	Python run 1	Python fastest	Julia run 1	Julia fastest
list and dispatch	3.60	3.12	0.43	0.19
CSV parse	0.06	0.06	0.49	0.17

tasks, and so is 3x to 8x faster than Julia. All of the Julia library code is written in Julia, but this of course is no help to an end user who only cares about application performance.

Julia is not yet able to cache generated native code, and so incurs a startup time of about two seconds to compile basic library functions. For some applications this latency is a barrier to deployment, and we plan to address it in the future.

5.2 Effectiveness of Specialization Heuristics

Given our implementation strategy, excessive compilation and corresponding memory use are potential performance concerns. In Figure 5-3 we present the number of method compilations performed on startup, and after running a test suite. From the second row of the table to the bottom, each of three specialization heuristics is successively enabled to determine its effect on compiler workload. In the last table row, each method is compiled just once.

The heuristics are able to elide about 12% of compilations. This is not a large fraction, but it is satisfying given that the heuristics can be computed easily, and only by manipulating types. On average, each method is compiled about 2.5 times.

Memory usage is not unreasonable for modern machines: on a 64-bit platform Julia uses about 50MB of memory on startup, and after loading several libraries and working for a while memory use tends to level off around 150-200MB. Pointer-heavy data structures consume a lot of space on 64-bit platforms. To mitigate this problem, we store ASTs and type information in a compact serialized format, and deserialize structures when the compiler needs them.

Figure 5-3: Number of methods compiled

	at startup	after test suite
no heuristics	396	2245
manual hints	379	2160
tuple widening	357	1996
vararg widening	355	1970
no specialization	267	766

5.3 Effectiveness of Type Inference

It is interesting to count compiled expressions for which a concrete type can be inferred. In some sense, this tells us "how close" Julia is to being statically typed, though in our case this is a property of both the language implementation and the standard library. In a run of our test suite, code was generated for 135375 expressions. Of those, 84127 (62%) had a type more specific than `Any`. Of those, 80874 (96%) had a concrete static type.

This suggests that use of dynamic typing is fairly popular, even though we try to avoid it to some extent in the standard library. Still, more than half of our code is well-typed. The numbers also suggest that, despite careful use of a rich lattice, typing tends to be an all-or-nothing affair. But, it is difficult to estimate the effect of the 4% abstractly-typed expressions on the other 96%, not to mention the potential utility of abstract inferred types in code that was not actually compiled.

These numbers are somewhat inaccurate, as they include dead code, and it may be the case that better-typed methods tend to be recompiled either more or less often, biasing the numbers.

5.4 Productivity

Our implementation of Julia consists of 11000 lines of C, 4000 lines of C++, and 3500 lines of Scheme (here we are not counting code in external libraries such as BLAS and LAPACK). Thus we have significantly less low-level code to maintain than most

scripting languages. Our standard library is roughly 25000 lines of Julia code. The standard library provides around 300 numerical functions of the sort found in all technical computing environments (arithmetic operators, reductions, sorting, etc.). We suspect that our library is one of the most compact implementations of this body of functionality.

At this time, every contributor except the core developers is a "new user" of Julia, having known of the language for no more than three months. Despite this, our function library has received several significant community contributions, and numerous smaller ones. We take this as encouraging evidence that Julia is productive and easy to learn.

Chapter 6

Related Work

There is a rich history of efforts to improve the performance of high-level dynamic languages. These can be broadly categorized into clever implementation tricks, which attempt to do the same work faster, and compiler-based techniques, which try to remove unnecessary operations from programs. This work is in the second category, which promises the greatest gains—the fastest way to do something is to avoid doing it altogether.

The operations we seek to remove are the run-time type manipulations that characterize dynamic languages: checking, applying, and removing type tags [29], and (typically) dispatching functions based on type tags. This can be seen as a special case of partial evaluation [25], wherein potentially *any* part of a program might be removed by reducing it to a constant before execution begins. There are some projects to develop partial evaluation as a practical tool for dynamic language execution (PyPy [8] is particularly relevant). The potential payoff there is quite high, as one may obtain not just a compiler for a particular language, but a compiler generator. However, this entails additional challenges that are not necessary if we are willing to focus on a single language. And for any given subject language, obtaining type information is the key goal for removing execution overhead.

Kaplan and Ullman produced an optimal algorithm for dynamic type inference [32], where the compiler attempts to guess what the run-time types of values will be. Inference is cast as a maximum-fixed-point dataflow problem. This is different from

type inference in the ML family of languages [40], where the compiler *must* be able to determine types, using an algorithm based on unification. Dynamic type inference is typically applied to an existing language with the hope of improving performance or safety. There are several examples in the Lisp world ([36], [6], [4], [29]), and in the world of object-oriented languages such as Self [11] and JavaScript [3].

Dynamic type inference schemes obey a correctness property that inferred types must subsume all possible run-time types. A trivial correct algorithm can be obtained by always returning a largest `Any` type. Thus an advantage of dynamic type inference is that the inference algorithm can be separate from the language specification. This allows the compiler to evolve and improve without changing the set of valid programs and without updating documentation.

On the other hand, dynamic type inference is easily defeated by systems with excessive type complexity. In these cases the alternate method of tracing [26] might work better. The idea of tracing is to record type information as a program actually runs instead of trying to guess it in advance. Tracing is well known for its use in modern JavaScript implementations [14]. Tracing is appealing as a "model-free" approach that can work for essentially any language, no matter how uncooperative its design. The corresponding disadvantages are that it cannot completely eliminate type checks (or "guards"), and that code must be interpreted with tracing overhead. When tracing fails to yield useful information, this overhead cannot be recovered and code might run slower than in the original system.

Yet another category of techniques involves altering a dynamic language by imposing a static type system, as in Typed Scheme [43] or an extended version of Dylan [37], or adding dynamic typing to a statically-typed language [28]. Gradual typing [42] allows programs to contain both statically-checked and run-time-checked components.

These techniques continue to be enhanced and applied to the current generation of popular scripting languages. DRuby [24] adds static type inference to Ruby. PRuby [23] extends this system to support highly dynamic constructs such as `eval`. RubyDust [2] uses a combination of constraint resolution and trace information to infer

types.

In this body of work it is commonly observed that dynamic language programs are not as dynamic as their authors might think: "We found that dynamic features are pervasive throughout the benchmarks and the libraries they include, but that most uses of these features are highly constrained..." [23]. In this sense, the designs of existing dynamic languages do not present a good trade-off. Much code is statically-typeable and could be executed more efficiently, but the language designs and implementations do not anticipate this fact. As Henry Baker observes of Common Lisp, "...the polymorphic type complexity of the Common Lisp library functions is mostly gratuitous, and both the efficiency of compiled code and the efficiency of the programmer could be increased by rationalizing this complexity."[4] Others have echoed these sentiments [9].

With type inference in mind from the beginning, unnecessary type complexity could be removed and a better overall system may result. Most importantly, the goal should not be just to make a dynamic language faster or safer, but to provide a more powerful language at the same time. For example, in applying type inference to a pre-existing language, it is typically useful to employ a finer type lattice than that provided by the original language (e.g. splitting integers into positive and negative subtypes). In some sense, this is unfortunate, as the extra expressiveness provided by these types is not available to the programmer for use in declarations, typecase statements, and the like. In light of the speedups possible with the techniques cited here, many common features of these languages can be seen as premature optimizations—for example employing only single dispatch and simple type systems (or, to be precise, "tag systems"). When we go to great lengths to make dynamic languages perform well, we should ask what else we can get for the same level of effort. Julia answers this question by employing more sophisticated types and dispatch rules that would make the language even slower without our compiler.

Julia uses generic functions and multiple dispatch as its primary abstraction mechanism. This feature famously appeared in the Common Lisp Object System (CLOS) [22] [7], and has been the subject of research languages such as Cecil [12] and Diesel

[13]. We find this style to be a good fit for mathematical programming, where operators are typically defined for many different combinations of arguments. In our context, an invaluable feature of generic functions is that they permit overloading for a variety of reasons: not just specializing behavior for subclasses, but for special cases that can be implemented more efficiently (e.g. dense double-precision arrays). We also find it a great simplification to support only generic functions, and not the usual mix including instance methods and class methods. Julia avoids some of the "overly permissive" features of CLOS [5] that get in the way of compiler optimizations.

The Dylan language [41] is close in spirit to the present work. It is also a dynamic language with multiple dispatch, designed with high performance as a goal. However, in Julia, automatic specialization is the primary source of type information, rather than user-supplied type declarations. We also explore the potential of automatic mechanisms for avoiding a combinatorial explosion of method specializations. Unlike Dylan, Julia does not make the distinction between generic functions and methods visible to the user—functions with the same name are simply combined into one generic function without restrictions. Julia method argument specializers also have a few more features useful for defining the highly polymorphic operators needed in technical computing. In Julia we also use our type machinery to introduce an extensible type promotion system, which allows operators to support all relevant combinations of arguments without requiring a large number of definitions for each new type.

The telescoping languages (TL) project [33] shares many of our goals, especially with respect to the domain of scientific computing. TL optimizes programs in MATLAB®and R by pre-processing those systems' standard libraries to generate a compiler that knows about commonly-used routines in order to optimize uses of them in subject programs. This project dramatically underscores the need for faster execution of these programs. The authors target library development as an especially important use case. The TL approach is fairly effective, but requires user annotations for peak performance. Julia's multimethods provide a way to collect type "annotations" in a less tedious manner—argument types specify dispatch behavior,

so they are part of the functionality of a program instead of being extraneous information. The off-line library analysis step in TL is subsumed by the Julia compiler's routine type analysis and its ability to cache type information to disk. Julia uses a mostly run-time compilation model, contrasted with TL's static generation of C or FORTRAN code. This enables an additional category of library implementation techniques where needed code can be generated on demand.

Of course, working on existing MATLAB®code is TL's greatest asset and greatest challenge. Other MATLAB®language compilers have been developed, such as FALCON [21]. The MathWorks now includes a JIT compiler in their product, and other JIT compilers such as MaJIC [1] have been described. Like Julia, the McVM [17] implementation of MATLAB®uses function specialization based on run-time types. However, their analysis by necessity includes details of MATLAB®'s data model that we prefer to leave out of the compiler, and they do not consider techniques for automatically reducing the amount of function specialization.

Some of these systems, including TL, have implemented advanced optimizations for vectorization [15] and array size analysis [16] that we have yet to explore. However, we feel Julia's open-source nature and powerful, "clean slate" base system would provide an excellent platform for such future work.

Chapter 7

Conclusion and Project Status

Julia was publicly announced in February 2012. Our goals and work so far seemed to strike a chord, as we have seen a significant community start to grow in the short time since then.

Julia is an open source project, with all code hosted on github [20]. We have over 450 mailing list subscribers, 1420 github followers, 170 forks, and more than 50 total contributors. Text editor support has been implemented for emacs, vim, and textmate. Github recognizes Julia as the language of source files ending in `.jl`, and can syntax highlight Julia code listings. We are currently #80 in github's language popularity ranking, up from #89 a month ago.

Several community projects are underway: two plotting packages, interfaces to arbitrary-precision arithmetic library GMP, bit arrays, linear programming, image processing, polynomials, GPU code generation, a statistics library, and a web-based interactive environment.

We hope Julia is part of a new generation of dynamic languages that not only run faster, but foster more cooperation between the programmer and compiler, pushing the standard of productivity ever higher.

Bibliography

[1] George Almasi and David Padua. Majic: A matlab just-in-time compiler. In Samuel Midkiff, Jos Moreira, Manish Gupta, Siddhartha Chatterjee, Jeanne Ferrante, Jan Prins, William Pugh, and Chau-Wen Tseng, editors, *Languages and Compilers for Parallel Computing*, volume 2017 of *Lecture Notes in Computer Science*, pages 68–81. Springer Berlin / Heidelberg, 2001.

[2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. *SIGPLAN Not.*, 46:459–472, January 2011.

[3] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In Andrew Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 733–733. Springer Berlin / Heidelberg, 2005.

[4] Henry G. Baker. The nimble type inferencer for common lisp-84. Technical report, Tech. Rept., Nimble Comp, 1990.

[5] Henry G. Baker. Clostrophobia: its etiology and treatment. *SIGPLAN OOPS Mess.*, 2(4):4–15, October 1991.

[6] Randall D. Beer. Preliminary report on a practical type inference system for common lisp. *SIGPLAN Lisp Pointers*, 1:5–11, June 1987.

[7] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *SIGPLAN Not.*, 23:1–142, September 1988.

[8] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[9] Rodney A. Brooks and Richard P. Gabriel. A critique of common lisp. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 1–8, New York, NY, USA, 1984. ACM.

[10] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[11] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self: a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24:49–70, September 1989.

[12] Craig Chambers. Object-oriented multi-methods in cecil. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 33–56, London, UK, 1992. Springer-Verlag.

[13] Craig Chambers. The diesel language specification and rationale: Version 0.1. February 2005.

[14] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 71–80, New York, NY, USA, 2009. ACM.

[15] Arun Chauhan and Ken Kennedy. Optimizing strategies for telescoping languages: procedure strength reduction and procedure vectorization. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, pages 92–101, New York, NY, USA, 2001. ACM.

[16] Arun Chauhan and Ken Kennedy. Slice-hoisting for array-size inference in matlab. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 495–508. Springer Berlin / Heidelberg, 2004.

[17] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing matlab through just-in-time specialization. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 46–65. Springer Berlin / Heidelberg, 2010.

[18] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[19] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer Berlin / Heidelberg, 1992.

[20] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[21] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. Falcon: A matlab interactive restructuring compiler. In Chua-Huang Huang, Ponnuswamy Sadayappan, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 269–288. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0014205.

[22] Linda DeMichiel and Richard Gabriel. The common lisp object system: An overview. In Jean Bzivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *ECOOP 87 European Conference on Object-Oriented Programming*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170. Springer Berlin / Heidelberg, 1987.

[23] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. *SIGPLAN Not.*, 44:283–300, October 2009.

[24] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[25] Yoshihiko Futamura. Partial evaluation of computation process: An approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12:381–391, 1999. 10.1023/A:1010095604496.

[26] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[27] Claude Gomez, editor. *Engineering and Scientific Computing With Scilab.* Birkhäuser, 1999.

[28] Fritz Henglein. Dynamic typing. In Bernd Krieg-Brckner, editor, *ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 233–253. Springer Berlin / Heidelberg, 1992.

[29] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, LFP '92, pages 205–215, New York, NY, USA, 1992. ACM.

[30] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5:299–314, 1996.

[31] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '86, pages 86–96, New York, NY, USA, 1986. ACM.

[32] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *J. ACM*, 27:128–145, January 1980.

[33] Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803 – 1826, 2001.

[34] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[35] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[36] Kwan-Liu Ma and Robert R. Kessler. Ticla type inference system for common lisp. *Software: Practice and Experience*, 20(6):593–623, 1990.

[37] Hannes Mehnert. Extending dylan's type system for better type inference and error detection. In *Proceedings of the 2010 international conference on Lisp*, ILC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[38] Markus Mohnen. A graphfree approach to dataflow analysis. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 185–213. Springer Berlin / Heidelberg, 2002.

[39] Malcolm Murphy. Octave: A free, high-level language for mathematics. *Linux J.*, 1997, July 1997.

[40] Robin and Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[41] Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[42] Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *ECOOP 2007 Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin / Heidelberg, 2007.

[43] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM.

[44] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011.