Abstraction in 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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2015

Author .....
Department of Electrical Engineering and Computer Science
January 9, 2015

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

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

Abstraction in Technical Computing

by

Jeffrey Werner Bezanson

Submitted to the Department of Electrical Engineering and Computer Science
on January 9, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Array-based programming environments are popular for scientific and technical computing. These systems consist of built-in function libraries paired with high-level languages for interaction. Although the libraries perform well, it is widely believed that scripting in these languages is necessarily slow, and that only heroic feats of engineering can at best partially ameliorate this problem.

In this thesis I argue that what is really needed is a more coherent structure for this functionality. To find one, we must ask what technical computing is really about. I suggest that this kind of programming is characterized by an emphasis on operator complexity and code specialization, and that a language can be designed to better fit these requirements.

The key idea is to integrate code *selection* with code *specialization*, using generic functions and data-flow type inference. Systems like these can suffer from inefficient compilation, or from uncertainty about what exactly to specialize on. I show that dispatch on structured type tags helps address these problems. The resulting language, Julia, achieves a Quine-style "explication by elimination" of many of the productive features technical computing users expect.

Thesis Supervisor: Alan Edelman
Title: Professor

Acknowledgments

Contents

Chapter 1

Introduction

Scientific computing has evolved from being essentially the only kind of computing that existed, to being a small part of how and why computers are used. Over this period of time, programming tools have advanced in terms of the abstractions and generalizations they are capable of. Many special-purpose languages have been subsumed by more powerful and general languages, much in the manner of improving scientific theories. How has this trend affected scientific computing? The surprising answer is: not as much as we would like. We see scientists and technical users generally either turning away from the "best" new programming languages, or else pushing them to their limits in unusual ways.

In fact, we have discovered at least *41* programming languages designed for technical computing since Fortran (figure 1-1). This is a high number by any standard. A reasonable hypothesis to explain this is that language design is the level of abstraction with the most flexibility (permitting *any* conceivable means of describing computations), so that is where you go when you need a lot of it. This is not so surprising when one reflects on the tradition of inventing new notation in mathematics and science.

The need for language-level flexibility is corroborated by the ways the technical computing domain uses general purpose languages. Effective scientific libraries extensively employ polymorphism, custom operators, and compile-time abstraction. Code generation approaches (writing a program that writes the needed program) are unusually common. Most of these

Matlab	Maple	Mathematica	SciPy
SciLab	IDL	R	Octave
S-PLUS	SAS	J	APL
Maxima	Mathcad	Axiom	Sage
Lush	Ch	LabView	O-Matrix
PV-WAVE	Igor Pro	OriginLab	FreeMat
Yorick	GAUSS	MuPad	Genius
SciRuby	Ox	Stata	JLab
Magma	Euler	Rlab	Speakeasy
GDL	Nickle	gretl	ana
Torch7			

Figure 1-1: 41 technical computing languages

techniques are presently fairly difficult to use, and so programmers working at this level give up the notational conveniences of the purpose-built languages above.

This thesis argues that the situation can be improved considerably by abstracting, with a new language, the essential challenges from the state of affairs we have described. Of course, a language is not the only thing the world of technical computing needs. Compiler techniques, library design, high-performance computational kernels, new algorithms, and approaches to parallelism might be more important than language design. However these sorts of technologies can usually be applied to multiple languages, as has happened in the C and Fortran language families. So in our view languages targeting this space need to dig deeper, at least temporarily forgetting about matrices and cache utilization, and look for bigger patterns.

In brief, our idea is to *integrate code selection and specialization*. The aforementioned flexibility requirement can be explained as sufficiently powerful means for *selecting* which piece of code to run. This notion subsumes method dispatch, function overloading, and potentially even branches. When such mechanisms are used in technical computing, there is nearly always a corresponding need to *specialize* code for specific cases to obtain performance. Polymorphic languages sometimes support some forms of specialization, but often only through a designated mechanism (e.g. templates), or deployed rather conservatively out

Mainstream PL	Technical computing
classes, single dispatch	complex operators
separate compilation	performance, inlining
parametric polymorphism	ad-hoc polymorphism, extensibility
static checking	experimental computing
modularity, encapsulation	large vocabularies
eliminating tags	self-describing data, acceptance of tags
data hiding	explicit memory layout

Figure 1-2: Priorities of mainstream object-oriented and functional programming language research and implementation compared to those of the technical computing domain.

of resource concerns. We intend to show that a kind of equillibrium point in the design space can be found by combining selection and specialization into essentially the same mechanism. This point is identified via subtyping theories and closure under data-flow operations.

1.1 The technical computing problem

Figure 1-2 compares the general design priorities of mainstream programming languages to those of technical computing languages. The priorities in each row are not necessarily opposites or even mutually exclusive, but rather are a matter of emphasis. For example, it is certainly possible to have both parametric and ad-hoc polymorphism within the same language, but syntax, recommended idioms, and the design of the standard library will tend to emphasize one or the other.

It is striking how different these priorities are. We believe these technical factors have contributed significantly to the persistence of specialized environments in this area.

1.2 Proposed solution

It is clear that any future scientific computing language will need to be able to match the performance of C, C++, and Fortran. To do that, it is almost certain that speculative

optimizations such as tracing [15] will not be sufficient — the language will need to be able to *prove* facts about types, or at least let the user request specific types. It is also clear that such a language must strive for maximum convenience, or else the split between performance languages and productivity languages will persist.

It is fair to say that two approaches to this problem are being tried: one is to design better statically-typed languages, and the other is to apply program analysis techniques to dynamically-typed languages. Static type systems are getting close to achieving the desired level of flexibility (as in Fortress [2] or Polyduce [9], for instance), but it is still too early to call a winner between these two approaches (if, indeed, there even needs to be a winner).

Here we will use a dynamically-typed approach. There are several reasons for this. First, we want to emphasize that insufficient static checking is most likely not the current limiting factor in scientific computing productivity. Second, some idioms in this domain appear to be inherently, or at least naturally, dynamically-typed (as we will explore in later chapters). Third, there has been a sense that "dynamic" or "scripting" language users do not want to hear about types: they are associated with verbosity and nuisance compiler errors. We hope to contribute an example of a language where types are useful, and not burdensome.

Efforts to analyze and optimize dynamically-typed programs generally make two assumptions: (1) we should work on analyzing *existing* popular languages, and (2) users of these languages don't want to use types. The first assumption makes practical sense. Convenience is hard to quantify, so using existing languages that have already been deemed convenient by popular opinion puts us on solid footing. Our work addresses the second assumption more directly. We point out that types [1] do not have to be used for static checking, and that using them for *code selection* and *code specialization* is particularly useful in technical computing. This perspective has not been explored thoroughly in the past.

When static analyses (often incorporating run-time information) are applied to dynamically-typed programs, it is typically possible to recover a significant amount of type information (TODO cite). What, then, can one do with this information? If the goal is performance,

[1] For an excellent discussion of the many meanings of the word "type" see [21].

various partial evaluations can be done: generating code without type checks, removing branches, type-specializing the storage of variables, and compile-time method lookup are all valuable and yield large real-world gains.

However, we claim that the amount of information that can be statically inferred exceeds most dynamic languages' capacity to exploit it. For example, if method calls are dispatched on the first argument, but the types of all arguments can be inferred, some power has been "left on the table" — we could have had multi-methods for little extra cost. In fact, method-at-a-time JIT compilers (TODO cite) can specialize method bodies on all arguments, and might use multiple dispatch *internally* to select implementations at run time (TODO cite a system that did this). This argument does not apply equally to statically-typed languages, since they cannot simply "switch" their functions to generic functions without significant consequences for type checking.

The "wasted power" problem applies to data structures as well. For example, static or run time analysis might reveal that a certain array can be represented as a native `Int32` array [4]. If this information is not reflected in the source language, then certain uses like passing data to native code become unnecessarily more complicated. And if one is going to implement homogeneous arrays anyway, why not let programmers request them?

Some levels of performance are difficult to reach with implicitly specialized code and data. Given the knowledge that an array contains only `Int32` data, we might want to go beyond essential optimizations like storing intermediate values in registers, and actually use different algorithms. For example, in Miller-Rabin primality testing, checking three "witness" values suffices for all 32-bit arguments, but up to seven values might be needed for 64-bit arguments (TODO cite). In cryptographic applications, exploiting this difference in an inner loop could bring significant benefits.

In demanding applications, selecting the right algorithm might not be enough, and we might need to automatically *generate* code to handle different situations. While these cases are relatively rare in most kinds of programming, they are remarkably common in technical computing. Code generation, also known as *staged programming*, raises several complexities:

- What is the input to the code generator?
- When and how is code generation invoked?
- How is the generated code incorporated into an application?

what to dispatch on? dispatch power has been extended in many ways, but there is no real limit to what somebody might want to dispatch on. so what to do? some sets of values are more robust under computation than others (closure properties). identify those sets using dataflow concerns.

say we have a method defined for integers, and also for the special cases "2" and "odd integers". a realistic implementation will group all of these under "integer", and ideally generate a couple branches to handle the other cases. we argue the concept of "integer" here is a more robust set, and so a more fundamental language concept.

somewhat counter-intuitively, dynamic dispatch can be good for performance since it permits invoking the most specialized possible method. static overloading can lead to calling a sub-optimal case when multiple overloads exist for the sake of performance.

The key ingredients:

1. Self-describing data model aware of memory layout
2. Type tags with nested structure
3. A fully-connected type tree
4. Dynamic multiple dispatch over all types
5. Dataflow type inference
6. Automatic code specialization

This list of features may appear somewhat ad-hoc. However, they turn out to be remarkably strongly coupled, and deeply constrained by our ultimate goal. Each of these features has appeared in some form before, but never in a way that fully solves the problems described here.

Challenges of this approach (why has this not been done before?)

1.3 Contributions

1 - an analysis of the nature of technical computing that suggests what sort of language would form a good base for it. it should emphasize complex operators and code generation/specialization.

2 - the idea of integrating specialization and selection, using multiple dispatch and semantic subtyping.

3 - an explication through elimination of technical computing language features

"One of the most fruitful techniques of language analysis is explication through elimination. The basic idea is that one explains a linguistic feature by showing how one could do without it." [26]

A main contribution of this thesis is the application of this approach to features of technical computing environments that have not been subject to such analysis before.

Chapter 2

Problems in technical computing

The original numerical computing language was Fortran, short for "Formula Translating System", released in 1957. Since those early days, scientists have dreamed of writing high-level, generic formulas and having them translated automatically into low-level, efficient machine code, tailored to the particular data types they need to apply the formulas to. Fortran made historic strides towards realization of this dream, and its dominance in so many areas of high-performance computing is a testament to its remarkable success.

The landscape of computing has changed dramatically over the years. Modern scientific computing environments such as MATLAB [25], R [20], Mathematica [23], Octave [27], Python (with NumPy) [32], and SciLab [18] have grown in popularity and fall under the general category known as *dynamic languages* or *dynamically typed languages*. In these programming languages, programmers write simple, high-level code without any mention of types like `int`, `float` or `double` that pervade *statically typed languages* such as C and Fortran.

How should a computer scientist approach this space? We might try to maximize performance. Or it is also easy to make unfounded assumptions about "what users want". But instead we should study the real world, see what is happening and figure out how to steer it in a better direction.

Hypothesis: people don't know what they want. It's also hard to predict what people

will want in the future. We need, in the words of Gerry Sussman, systems adaptable to uses not imagined by their designers.

2.1 What is a technical computing environment?

This question has not really been answered.

Views of this are strongly shaped by what systems happen to exist, and what people were exposed to as they learned to program.

Technical computing software has been designed haphazardly. Each system is a pile of features taken without argument.

Some languages provide a "convenient" experience that is qualitatively different from "inconvenient" languages. We believe this can be made somewhat precise. A large part of it is reducing the amount you need to know about any given piece of functionality in order to use it.

These systems are function-oriented, typically providing a rather large number of functions and a much smaller number of data types.

These functions, and the ways they are used, have a particular character of being "manifest": one can just call them, interactively, and see what they do. This notion includes the following features:

1. Performing fairly large tasks in one function
2. Minimal "set up" work to obtain suitable arguments
3. No surrounding declarations required
4. Permissiveness in accepting many data types and attempting to automatically handle as many cases as possible
5. Providing multiple related algorithms or behaviors under a single name

Language design choices affect the ability to provide this user experience (though the first two items are also related to library design). Informally, in order to provide the desired

experience a language needs to be able to assign a meaning to a brief and isolated piece of code such as `sin(x)`. This leads directly to making declarations and annotations optional, eliding administrative tasks like memory management, and leaving information implicit (for example the definition scopes and types of the identifiers `sin` and `x`). These characteristics are strongly associated with the Lisp tradition of dynamically-typed, garbage collected languages with interactive REPLs.

However, there are subtle cultural differences. A case in point is the MATLAB `mldivide`, or backslash, operator [24]. By writing only `A\B`, the user can solve square, over- or under-determined linear systems that are dense or sparse, for multiple data types. The arguments can even be scalars, in which case simple division is performed. In short, a rather large amount of linear algebra software is accessed via a single character! This contrasts with the software engineering tradition, where clarifying programmer intent would likely be considered more important. (TODO cite if possible?) Even the Lisp tradition, which originated most of the convenience features enumerated above, has sought to separate functionality into small pieces. For example Scheme provides separate functions `list-ref` and `vector-ref` [1] for indexing lists and vectors.

2.2 Why dynamic typing?

Mathematical abstractions often frustrate our attempts to represent them on a computer. For example, mathematicians can move instinctively between isomorphic objects such as scalars and 1-by-1 matrices, but most programming languages would prefer to represent scalars and matrices quite differently. A system might be able to convert automatically from one to the other, but it cannot always know which one we *meant*.

2.2.1 Mismatches with mathematical abstractions

Programs in general, deal with values of widely varying disjoint types: functions, numbers, lists, network sockets, etc. Type systems are good at sorting out values of these different

types. However, in mathematical code most values are numbers or number-like. Numerical properties (such as positive, negative, even, odd, greater than one, etc.) are what matter, and these are highly dynamic.

The classic example is square root (`sqrt`), whose result is complex for negative arguments. Including a number's sign in its type is a possibility, but this quickly gets out of hand. When computing eigenvalues, for example, the key property is matrix symmetry. Linear algebra provides many more examples where algorithm and data type changes arise from subtle properties. These will be discussed in more detail in section 5.2.

We must immediately acknowledge that static type systems provide mechanisms for dealing with "dynamic" variations like these.

multiple common features underlie mathematical objects of different types (e.g. numbers, sets, matrices). in some cases it makes sense to consider numbers and matrices as the same kind of thing, and in other cases it doesn't matter. A given type system is likely not to have anticipated the particular common features that matter to your program, making it more difficult to express an idea. A concrete example is the matlab fragment if condition idx = ':' else idx = 1 end where we want to select either an entire dimension or the first position alone. The ':' and 1 are both indexes in this context, though they would be of disjoint types in most programming languages.

2.2.2 Operational reasoning

people tend to think about programs operationally, i.e. what it *does* when it runs. for example writing if false code end the code does not "occur" and therefore does not need to be valid

there is less to learn. with static languages you have to learn what happens at both compile time and run time, when only run time really matters.

there is a desire to parameterize as much as possible. functions accept parameters, so function calling ought to be sufficient to express any desired parameterization.

2.2.3 I/O

inevitably there is a need to refer to a datatype at run time. the best example is file I/O, where you might want to say "read 500 double-precision numbers from file X". in static languages the syntax and identifiers used to specify such run-time types must be different from those used to specify static types. in C you see defined constants such as `DATATYPE_DOUBLE`.

2.2.4 Flat metadata hierarchy

static types are approximations of dynamic types, so languages with static types inevitably assign two types to a location (both a static type and a dynamic type) where one would do. in some languages, like C++, the desire for performance or ease of implementation leads the compiler to make some decisions based on static types. this is confusing. if type declarations can be omitted, as in a type-inferred language, the situation is even worse since the static type of a value might not be apparent.

2.3 The software stack is too complex

Collapsing abstraction layers

- for numerical debuggability in particular - debuggability in general - no time spent worrying about binding time - you will build a dynamic dispatch layer anyway, so build it in

How much of the past 30 years of handwritten Matlab internals can be autogenerated with a compiler? (A lot)

Can now get past traditional classifications of language boundaries - interpreted vs. compiled, high-level vs. low-level, procedural vs. imperative vs. functional, etc.

language performance psychology: if your language doesn't directly support efficient machine data types, users will rewrite their code in C in order to get them, and then be happy with the result (though not with the process).

so julia is an all-levels language

2.4 Code generation

- tensor contraction engine - Firedrake, pyop2 (vs. c++ libmesh) - JuMP vs. python puLP and pyomo - want to be able to pass functions, not C++ code as strings - FFTW - erfinv and horner

2.4.1 A compiler for every problem

2.5 Bad tradeoffs in current designs

- Ducking the issue of typing altogether - why this isn't possible IRL

Case in point: Python and NumPy.

NumPy tried to work within the confines of Python, but has more or less failed for technical and political reasons. (Technical: hard to work with types in language that deliberately tries to obscure them from the user. Political: unwillingness to fix broken package distribution system.) Consequences: Numba and Numba_lang, Anaconda. Essentially reinventing types in "Python".

This phenomenon is increasingly noticed in other domains, particularly JavaScript and web programming. Modern JavaScript implementations are quite fast, but Google's Dart language is based on the premise that we could have a web language that is even faster, and offers more productivity as well. How so? Because Dart's designers observed that JavaScript programmers in practice often write code that could be defined using traditional OO classes, but the language does not support them.

dictionaries for everything (python, js) is the wrong default. almost every type somebody wants has a fixed number of fields with fixed types.

Python is often described as a good glue language. This means it is effectively used as an interface standard, a kind of extended C ABI that makes it easier for libraries to interoperate, and easier for users to access those libraries. Something as straightforward as providing a standard N-dimensional numeric array class (NumPy), which does not exist at the level of

C, goes a long way.

However, we wish to point out that in this picture, Python is not doing as much work as it might first appear. Python does not make it easier to implement the functionality inside NumPy, or other "under the hood" scientific libraries. In many cases it creates more work, through the need to write wrappers and interfaces for native code.

Merely being "dynamic" (e.g. Python) should not be considered the gold standard of flexibility. Although these systems permit tricks that can solve otherwise difficult programming problems, this is not always the kind of flexibility that is needed. When faced with the need to describe many functions with elaborate behavior and many cases, one does not primarily need permissiveness, but rather powerful and descriptive organizing principles.

2.5.1 What needs to be built in?

On "built-in", why built-in is bad. Built-in-ness often conflates two aspects:

1. A feature being readily available and agreed-on by all language users

2. A feature tightly coupled to the rest of the system

(2) implies (1), but not the other way around. (2) is the only technically interesting item, since the other can be addressed e.g. just by including a library in the standard software distribution. Many technical computing languages have done a large amount of (2) while justifying it with point (1).

While a large part of our motivation is to move more decisions and functionality into libraries, it is equally important to identify what *must* be part of a language for the system to be successful. We believe that large amounts of functionality can be provided by add-ons, but that certain key features absolutely cannot be. Past failures to properly classify features this way have caused a lot of undue pain.

First, performance cannot be an add-on. If some users have a fast version of a language and others have a slow version (with the difference being an order of magnitude or more), library writers cannot be sure whether users will find their code fast enough to be useful.

How are we to teach people to program in the language? Courses on MATLAB programming emphasize vectorized code, but if not all implementations had this performance characteristic the curriculum would become confused.

Psychologically, it may be difficult to accept a "non standard" extension that changes a language so fundamentally. There is a nagging, though perhaps totally unfounded, perception that something subtle may break. If indeed a bug arises due to the use of such an extension, a user is likely to conclude that the extension is dangerous or broken and stop using it. If, on the other hand, a bug arises due to a language's standard optimizing compiler, the user will simply file a bug report, then find a way to work around the problem.

Adding a JIT compiler to a language also requires acceptance of detriments like compilation pauses and pages with RWX permissions. In some cases this may lead to use of the extension being disallowed, perhaps for security reasons.

Type systems similarly fail when provided as optional extensions. Library writers face the same kinds of problems as with performance add-ons. Should I use type annotations in my library?

Dynamic dispatch mechanisms also make especially poor add-ons. Of course, every program makes decisions at run time, and so implements its own "dispatch" to some extent. But these behaviors are inextensible; if language users do not agree on a reusable dispatch framework their code will not be composable.

2.6 Social phenomena

Programming languages are observed to have strong network effects, and the difficulty of getting new languages adopted is well known. However based on [socio PLT paper] we believe this doesn't have to be the case. The formula of improving or redesigning general-purposes languages to be more appealing to domain experts might solve the problem. That way the new system has immediate appeal for at least some users, without the worry that a different tool will be needed as soon as requirements change slightly.

barrier to contributing

Software business is based on imposing restrictions

Debates about what abstractions mean – AbstractMatrix we thought we knew what it meant, but what about something like SymTridiagonal? It can implement most of what a dense matrix has, but it can't obey every invariant.

PackedQR

Chapter 3

Available technology

3.1 The language design space

It is helpful to begin with a rather coarse classification of programming languages, according to how expressive their type systems are, and whether their type systems are dynamic or static:

	More types	Fewer types
Dynamic	Dylan, Julia	Scheme, Python, MATLAB
Static	ML, Haskell, Scala	C

The lower right corner tend to contain older languages. The upper right corner contains many popular "dynamic" languages. The lower left corner contains many modern languages resulting from research on static type systems. We are most interested in the upper left corner, which is notable for being rather sparsely populated. It has been generally believed that dynamic languages do not "need" types, or that there is no point in talking about types if they are not going to be checked at compile time. These views have some merit, but as a result the top-left corner of this design space has been seriously under-explored.

3.1.1 Classes vs. functions

3.1.2 Separate compilation vs. performance

Writing the signature of a generic method that needs to be separately compiled, as in Java for example, can be a difficult exercise. The programmer must use the type system to write down sufficient conditions on all arguments. (TODO example from [16]) If, however, we are going to specialize the method, the compiler can analyze it using actual types from call sites, and see for itself whether the method works in each case (this is how C++ templates are type-checked; they are analyzed again for each instantiation).

It is quite interesting that performance and ease-of-use pull this design decision in the same direction.

3.1.3 Parametric vs. ad-hoc polymorphism

The idea of specialization unites parametric and ad-hoc polymorphism. Beginning with a parametrically-polymorphic function, one can imagine a compiler specializing it for various cases, i.e. certain concrete argument values. These specializations could be stored in a lookup table, for use either at compile time or at run time.

Next we can imagine that the compiler might not optimally specialize certain definitions, and that the programmer would like to provide hints or hand-written implementations to speed up special cases. For example, imagine a function that traverses a generic array. A compiler-generated specialization might inline a specific array types's indexing operations, but a human might further realize that the loop order should be switched for certain arrays types based on their storage order.

However, if we are going to let a human specialize a function for performance, we might as well allow them to specialize it for some other reason, including entirely different behavior. But at this point separate ad-hoc polymorphism is no longer necessary; we are using a single overloading feature for everything.

Parametric polymorphism describes code that works for any object precisely because

it does not do anything meaningful to the object, for example the identity function. In contrast, programming with tagged data (e.g. symbolic expression systems, XML) permits code to work for any object because every object has the same structure, allowing meaningful operations.

3.1.4 Static checking vs. flexibility

3.1.5 Modularity vs. large vocabularies

In the context of software engineering, modularity is a primary concern. To build large systems, separate components must be isolated to some degree. Reasons for this include simple concerns like avoiding name conflicts, and a desire to separate interface from implementation to allow a component to change without affecting the rest of the system.

Modularity is sometimes taken to an extreme, and one will see fully qualified names like `org.jboss.annotation.ejb.cache.simple.CacheConfig` (selected from the JBOSS Java APIs).

Technical computing languages have often avoided and discouraged such designs. For example MATLAB for most of its history supported only a single namespace, which comes pre-populated with thousands of functions.

3.1.6 Eliminating vs. embracing tagged data

3.1.7 Data hiding vs. explicit memory layout

Examples of CSC and CSR sparse representation. In one sense this is a perfect example of an interface with multiple implementations, and therefore a good use case for object-oriented programming. However in the technical computing world, *hiding* this difference in representation is not usually considered desirable. Clearly a sparse matrix class cannot contain all functions of matrices that users might want to compute. Yet when new functions are added, the programmer needs and wants to exploit representation details (CSC or CSR)

for performance. The performance differences involved here are quite significant (TODO cite).

The loss of encapsulation due to multi-methods weighed in [5] is less of a problem for technical computing, and in some cases even advantageous.

3.1.8 Dynamic dispatch is key

It would be unpleasant if every piece of every program we wrote were forced to do only one specific task. Every time we wanted to do something slightly different, we'd have to write a different program. But if a language allows the same program element to do different things at different times, we can write whole classes of programs at once. This kind of capability is one of the main reasons *object-oriented* programming is popular: it provides a way to automatically select different behaviors according to some structured criteria.

In class-based OO there is essentially *no way* to create an operation that dispatches on existing types (the expression problem). This clearly does not match technical computing, where most programs deal with the same few types (e.g. number, array), and might sensibly want to write new operations that dispatch on them.

We use the non-standard term "criteria" deliberately, in order to clarify our point of view, which is independent of any particular object system.

The sophistication of the available "selection criteria" account for a large part of the perceived "power" or leverage provided by a language. In fact it is possible to illustrate a hierarchy of such mechanisms. As an example, consider a simple simulation, and how it can be written under a series of increasingly powerful paradigms. First, written-out imperative code:

```
while running
    for a in animals
        b = nearby_animal(a)
        if a isa Rabbit
            if b isa Wolf then run(a)
```

```
            if b isa Rabbit then mate(a,b)
        else if a isa Wolf
            if b isa Rabbit then eat(a,b)
            if b isa Wolf then follow(a,b)
        end
    end
end
```

We can see how this would get tedious as we add more kinds of animals and more behaviors. Another problem is that the animal behavior is implemented directly inside the control loop, so it is hard to see what parts are simulation control logic and what parts are animal behavior. Adding a simple object system leads to a nicer implementation [1]:

```
class Rabbit
    method encounter(b)
        if b isa Wolf then run()
        if b isa Rabbit then mate(b)
    end
end

class Wolf
    method encounter(b)
        if b isa Rabbit then eat(b)
        if b isa Wolf then follow(b)
    end
end

while running
```

[1] A perennial problem with simple examples is that better implementations often make the code longer.

```
    for a in animals
        b = nearby_animal(a)
        a.encounter(b)
    end
end
```

Here all of the simulation's animal behavior has been essentially compressed into a single program point: `a.encounter(b)` leads to all of the behavior by selecting an implementation based on the first argument, `a`. This kind of criterion is essentially indexed lookup; we can imagine that `a` is simply an integer index into a table of operations.

The next enhancement to "selection criteria" adds a hierarchy of behaviors, to provide further opportunities to avoid repetition:

```
abstract class Animal
    method nearby()
        # search within some radius
    end
end

class Rabbit <: Animal
    method encounter(b: Animal)
        if b isa Wolf then run()
        if b isa Rabbit then mate(b)
    end
end

class Wolf <: Animal
    method encounter(b: Animal)
        if b isa Rabbit then eat(b)
```

```
        if b isa Wolf then follow(b)
    end
end

while running
    for a in animals
        b = a.nearby()
        a.encounter(b)
    end
end
```

We are still essentially doing table lookup, but the tables have more structure: every `Animal` has the `nearby` method, and can inherit a general-purpose implementation.

This brings us roughly to the level of most popular object-oriented languages. But in this example still more can be done. Notice that in the first step to objects we replaced one level of `if` statements with method lookup. However, inside of these methods a structured set of `if` statements remains. We can replace these by adding another level of dispatch.

```
class Rabbit <: Animal
    method encounter(b: Wolf) = run()
    method encounter(b: Rabbit) = mate(b)
end

class Wolf <: Animal
    method encounter(b: Rabbit) = eat(b)
    method encounter(b: Wolf) = follow(b)
end
```

We now have a *double dispatch* system, where a method call uses two lookups, first on the first argument and then on the second argument.

This syntax might be considered a bit nicer, but the design clearly begs a question: why is $n = 2$ special? It isn't, and we could clearly consider even more method arguments as part of dispatch. But at that point, why is the first argument special? Why separate methods in a special way based on the first argument? It seems arbitrary, and indeed we can remove the special treatment:

```
abstract class Animal
end

class Rabbit <: Animal
end

class Wolf <: Animal
end

nearby(a: Animal) = # search
encounter(a: Rabbit, b: Wolf) = run(a)
encounter(a: Rabbit, b: Rabbit) = mate(a,b)
encounter(a: Wolf, b: Rabbit) = eat(a, b)
encounter(a: Wolf, b: Wolf) = follow(a, b)

while running
    for a in animals
        b = nearby(a)
        encounter(a, b)
    end
end
```

Here we made two major changes: the methods have been moved "outside" of any classes, and all arguments are listed explicitly. This change has fairly significant implications. Since

methods no longer need to be "inside" classes, there is no syntactic limit to where definitions may appear. Now it is easier to add new methods after a class has been defined. Methods also now naturally operate on combinations of objects, not single objects.

The shift to thinking about combinations of objects is fairly revolutionary. Many interesting properties only apply to combinations of objects, and not individuals. We are also now free to think of more exotic kinds of combinations.

We can define a method on *any number* of objects:

```
encounter(ws: Wolf...) = pack(ws)

encounter{T<:Animal}(a: T, b: T) = mate(a, b)
```

3.2 Multiple dispatch

CHART of different multiple dispatches. classify them

Why "just overloading" is not enough. You intercept every operation and rewrite it, kind of an escape hatch for a language you don't like. If somebody else also tries to do this, there won't necessarily be any coherence.

A helpful way to classify languages with some kind of generic programming support is to look at which language constructs are generic. For example, in C++ the syntax `object.method(x)` is generic: a programmer can get it to do different things by supplying different values for `object`. The C++ syntax `f(a,b)` or `a+b` is usually not generic, but can be overloaded by supplying definitions for different argument types. However, this overloading only uses compile-time types (no run-time information), and so is essentially a form of renaming — it can be implemented by renaming each definition with a unique name, and replacing overloaded calls with an appropriate name based on compile-time types. This is a much weaker form of generic programming, since different behaviors cannot be obtained by passing different values at run time.

For this reason, C++ programs are not fully generic: it is difficult to substitute new behaviors for every part of a program. Some languages take generic programming much fur-

ther. For example, in MATLAB, *every* function is effectively generic, and can be overloaded by new classes. This enables a programmer to "intercept" every function call (and therefore, essentially everything a program does)

foo

3.2.1 Example: lattices

This example will illustrate possible benefits of multiple dispatch and dynamic typing for mathematical computing. The benefits are not absolute, but notational and semantic: they involve code size and clarity, and the extent to which the entities provided by the language match a mental model of the domain.

A *lattice* is an algebraic structure where some pairs of elements satisfy a reflexive, anti-symmetric, and transitive relation $\leq$. For purposes of this example, we will consider lattices that have a greatest, or *top*, element ($\top$), and a least, or *bottom* element ($\bot$). When working with lattices one often wants to compute a least upper bound, or *join* ($\sqcup$), or a greatest lower bound, or *meet* ($\sqcap$).

Several interesting concerns arise when modeling lattices in a programming language. First, the structure is very general, and so admits implementations for many different kinds of elements. We want to write code using the operators $\leq$, $\sqcup$, and $\sqcap$, and have it apply to any kind of lattice. Therefore some kind of overloading or object-oriented programming is desirable. Second, some properties apply to all lattices, and we would like to avoid implementing them repeatedly.

Using "duck typing", the problem of modeling an abstraction like lattices disappears almost entirely. One may simply define methods for $\leq$, $\sqcup$, and $\sqcap$ at any time, for any type, and that type will function as a lattice. That is certainly convenient, but it also fails to provide any reusable functionality for those defining lattices.

Figure 3.2.1 shows a small Julia library for lattices. It defines an abstract class `LatticeElement` that may be subclassed by objects that will be used primarily as elements of some lattice. The library also provides standard `LatticeElement` provides some useful default method

```
abstract LatticeElement

<=(x::LatticeElement, y::LatticeElement) = x===y
==(x::LatticeElement, y::LatticeElement) = x<=y && y<=x
< (x::LatticeElement, y::LatticeElement) = x<=y && !(y<=x)

immutable TopElement <: LatticeElement; end
immutable BotElement <: LatticeElement; end

const ⊤ = TopElement()
const ⊥ = BotElement()

<=(::BotElement, ::TopElement) = true
<=(::BotElement, ::LatticeElement) = true
<=(::LatticeElement, ::TopElement) = true

⊔(x::LatticeElement, y::LatticeElement) =  # join
    (x <= y ? y : y <= x ? x : ⊤)

⊓(x::LatticeElement, y::LatticeElement) =  # meet
    (x <= y ? x : y <= x ? y : ⊥)
```

definitions.

3.2.2 Symbolic programming

There has always been an essential divide between "numeric" and "symbolic" languages in the world of technical computing. To many people the distinction is fundamental, and we should happily live with both kinds of languages. But if one insists on an answer as to which approach is the right one, then the answer is: symbolic.

Systems based on symbolic rewrite rules arguably occupy a further tier of dispatch sophistication. In these systems, you can dispatch on essentially anything, including arbitrary values and structures. These systems are typically powerful enough to concisely define the kinds of behaviors we are interested in.

However, symbolic programming lacks data abstraction: the concrete representations of values are exposed to the dispatch system (e.g. there is no difference between being a list and being something represented as a list).

3.2.3 Predicate dispatch

Predicate dispatch is a powerful object-oriented mechanism that allows methods to be selected based on arbitrary predicates (TODO cite). It is, in some sense, the most powerful *possible* dispatch system, since any computation may be done as part of method selection. Since a predicate denotes a set (the set of values for which it is true), it also denotes a set-theoretic type. Some type systems of this kind, notably that of Common Lisp (TODO cite), have actually included predicates as types. However, such a type system is obviously undecidable, since it requires computing the predicates themselves or, even worse, computing predicate implication. [2]

[2]Many type systems involving bounded quantification, such as system $F_{<:}$, are already undecidable [28]. However, they seem to terminate for most practical programs, and also admit minor variations that yield decidable systems [10]. It is fair to say they are "just barely" undecidable, while predicates are "very" undecidable.

For a language that is willing to do run-time type checks anyway, the undecidability of predicate dispatch is not a problem. Interestingly, it can also pose no problem for *static* type systems that wish to prove that every call site has an applicable method. Even without evaluating predicates, one can prove that the available methods are exhaustive (e.g. methods for both p and $\neg p$ exist). In contrast, and most relevant to this thesis, predicate types *are* a problem for code *specialization*. Static method lookup would require evaluating the predicates, and optimal code generation would require understanding something about what the predicates mean. One approach would be to include a list of satisfied predicates in a type. However, to the extent such a system is decidable, it is essentially equivalent to multiple inheritance. Another approach would be to separate predicates into a second "level" of the type system. The compiler would combine methods with the same "first level" type, and then generate branches to evaluate the predicates. Such a system would be very useful, and could be combined with a language like Julia or, indeed, most object-oriented languages. However this comes at the expense of conceding that predicates are not "real" types.

In considering the problems of predicate dispatch for code specialization, we seem to be up against a fundamental obstacle: some sets of values are simply more robust under evaluation than others. Programs that map integers to integers abound, but programs that map, say, even integers to even integers are rare to the point of irrelevance.

3.3 Domain theory

In the 1960s Dana Scott asked how to assign meanings to programs, which otherwise just appear to be lists of symbols. For example, given a program computing the factorial function, we want a process by which we can assign the meaning "factorial" to it.

This is about modeling the behavior of a program without running it.

The idea of analyzing computer programs began in earnest in the 1960s with Dana Scott's invention of domain theory. A "domain" in this theory is a partial order of sets of values that a program might manipulate. Domain theory models computation as follows:

a program starts with no information, the lowest point in the partial order ("bottom"). Computation steps accumulate information, gradually moving higher through the order. The advantage of this model, in essence, is that it provides a way to think about the meaning of a program without running the program. Even the "result" of a non-terminating program has a representation — the bottom element. Other elements of the partial order might refer to intermediate results.

Domain theory gave rise to the study of denotational semantics and the design of type systems. However, the original theory is quite general and invites us to invent any domains we wish for any sort of language. For example, given a program that outputs an integer, we might decide that we only care whether this integer is even or odd. Then our posets are the even and odd integers, and we will classify operations in the program according to whether they are evenness-preserving and so on. It should be clear that this sort of analysis, while clearly related to type systems, is fairly different from what most programmers think of as type checking.

3.3.1 Monotonicity

Functions of types are not monotonic. `isa(2,3)` is `false`, but `isa(2,Int)` is `true`. This provides one way to separate "real types" from set-valued terms: if there is a way to interpret execution over types such that all functions are monotonic in the (alleged) type lattice, then we are really dealing with types. One way to do this is to specify a genuinely multi-valued term language like $\lambda_{\aleph}$ [17]. In this approach a term like `isa(2,Int)` is interpreted as a union of all results obtained by substituting all possible single values for each sub-term. One cannot help but be impressed by such a language, though it seems prohibitively difficult to actually implement. We opt instead for a simpler approach where `Int` is interpreted as `Type{Int}` (in a near-borrowing of notation from [17] and [6]), and multi-valued terms exist only statically within a separate maximum-fixed-point evaluator.

3.4 Subtyping

Something about semantic subtyping and type systems for processing XML. XML at first seems unrelated to numerical computing, and indeed it was quite a while before we discovered these papers and noticed the overlap. However if one substitutes "symbolic expression" for "XML document", the similarity becomes clearer.

3.5 Dynamic type inference

This is exciting because the type system can do useful computational work for you, instead of only checking the work done by the rest of the program.

Data flow analysis has to actually work

Taking things out of the language that break dataflow analysis

Local reasoning

3.5.1 Objections to dynamic typing

it may be that the "power" of a language is equal to the complexity of the criteria used by the language's run-time dispatch mechanisms.

pre-OO: pointer indirection OO: single dispatch, class hierarchies

Haskell: without typeclasses, a beautiful language, but one that nobody would use.

I claim that compile-time abstractions do not count. The problem is that at some point you have to *actually run the program*. Specifying the behavior of a program is hard; this is where we need help from the language.

Scripting languages are often defended using the observation that most code is not performance-critical. However, this fact does not mean that it's ok for a language design to ignore performance, nor does it mean that performance should be the default priority of every line of code. Rather it means that the default should be convenience and safety, with a path to performance easily in reach for when it's needed.

Chapter 4

The Julia approach

4.1 Core calculus and data model

Julia is based on an untyped lambda calculus augmented with generic functions, tagged data values, and mutable cells.

$$
\begin{array}{rcll}
e & ::= & x & \text{(variable)} \\
& & \mid 0 \mid 1 \mid \cdots & \text{(constant)} \\
& & \mid \texttt{new}(e_{tag}, e_1, e_2, \cdots) & \text{(data constructor)} \\
& & \mid e_1.e_2 & \text{(projection)} \\
& & \mid e_1.e_2 = e_3 & \text{(assignment)} \\
& & \mid \texttt{if } e_1\ e_2\ e_3 & \text{(conditional)} \\
& & \mid e_1(e_2) & \text{(application)} \\
& & \mid e_1; e_2 & \text{(sequencing)} \\
& & \mid \texttt{function } x_{name}\ e_{type}\ (x_1, x_2, \cdots)\ e_{body} & \text{(method definition)}
\end{array}
$$

Tags are a specific subset of data values generated by `new` where the first argument is a special built-in tag `Tag`. Though `new` is an important part of the core calculus, it is only available in the source language in a syntactically-restricted form that provides stronger data abstraction.

Constants are pre-built tagged values.

Types are a superset of tags that includes values generated by the special tags `Abstract`, `Union`, and `UnionAll`, plus the special values `Any` and `Bottom`:

$$\begin{aligned} \mathit{type} \;::=\; & \texttt{Bottom} \mid \mathit{abstract} \mid \mathit{var} \\ & \mid \texttt{Union}\ \mathit{type}\ \mathit{type} \\ & \mid \texttt{UnionAll}\ \mathit{type}\texttt{<:}\mathit{var}\texttt{<:}\mathit{type}\ \mathit{type} \\ & \mid \texttt{Tag}\ x_{\mathit{name}}\ \mathit{abstract}_{\mathit{super}}\ \mathit{value}* \\ \mathit{abstract} \;::=\; & \texttt{Any} \mid \texttt{Abstract}\ x_{\mathit{name}}\ \mathit{abstract}_{\mathit{super}}\ \mathit{value}* \\ & \mid \texttt{Abstract Tuple Any}\ \mathit{type}*\ \mathit{type}\ldots \end{aligned}$$

The last item is the special abstract varargs tuple type.

The language implicitly maps tags to data descriptions, and ensures that the data part of a tagged value always conforms to the tag's description. Mappings from tags to data descriptions are established by special type declaration syntax. Data descriptions have the following grammar:

$$\mathit{data} \;::=\; \mathit{bit}^n \mid \mathit{ref} \mid \mathit{data}*$$

where bit^n represents a string of n bits, and ref represents a reference to a tagged data value. Data may be declared mutable, in which case its representation is implicitly wrapped in a mutable cell. A built-in primitive equality operator `===` is provided, based on *egal* [3] (mutable objects are compared by address, and immutable objects are compared by directly comparing both the tag and data parts bit-for-bit, and recurring through references to other immutable objects).

Functions are generally applied to more than one argument. In the application syntax $e_1(e_2)$, e_2 is an implicitly-constructed tuple of all arguments.

We use the keyword `function` for method definitions for the sake of familiarity, though `method` is arguably more appropriate. Method definitions subsume lambda expressions. Each method definition modifies a generic function named by the argument x_{name}. The function to extend is specified by name rather than by value in order to make it easier to syntactically restrict where functions can be extended. This, in turn, allows the language to specify

when new method definitions take effect, providing useful windows of time within which methods do not change, allowing programs to be optimized more effectively (and hopefully discouraging abusive and confusing run-time method replacements).

The equivalent of ordinary lambda expressions can be obtained by introducing a unique local name and defining a single method on it.

Mutable lexical bindings are provided by the usual translation to operations on mutable cells.

The signature, or specializer, of a method is obtained by evaluating e_{type}, which must result in a type value as defined above. A method has n formal argument names x_i. The specializer must be a subtype of the varargs tuple type of length n. When a method is called, its formal argument names are bound to corresponding elements of the argument tuple. If the specializer is a vararg type, then the last argument name is bound to a tuple of all trailing arguments.

4.1.1 A note on static typing

4.2 Type system

Our goal is to design a type system useful for describing method applicability, and (similarly) for describing classes of values for which to specialize code. Set-theoretic types are a natural basis for such a system. A set-theoretic type is a symbolic expression that denotes a set of values. In our case, these correspond to the sets of values methods are intended to apply to, or the sets of values supported by compiler-generated method specializations. Since set theory is widely understood, the use of such types tends to be intuitive.

These types are less coupled to the languages they are used with, since one may design a value domain and set relations within it without yet considering how types relate to program terms (TODO cite Castagna). Since our goals only include performance and expressiveness, we simply skip the later steps for now, and do not address how to type-check terms (or, indeed, the question of whether checking is even possible).

To avoid the dual traps of "wasted power" and divergence, the system we use must have a decidable subtype relation, and must be closed under data-flow operations (meet, join, and widen). It must also lend itself to a reasonable definition of specificity, so that methods can be ordered automatically (a necessary property for extensibility). These requirements are fairly strict, but still admit many possible designs. The one we present here is aimed at providing the minimum level of sophistication needed to yield a language that feels "powerful" to most modern programmers. Beginning with the simplest possible system, we added features as needed either to satsify the aforementioned closure properties, or to allow us to write method definitions that seemed particularly useful (as it turns out, these two considerations lead to essentially the same features). The presentation that follows will partially reproduce the order of this design process.

We will define our types by formally describing their denotations as sets. We use the notation $[\![T]\!]$ for the set denotation of type expression T. Concrete language syntax and terminal symbols of the type expression grammar are written in typewriter font, and meta-symbols are written in mathematical italic. First there is a universal type `Any`, an empty type `Bottom`, and a partial order $\leq$:

$$[\![\texttt{Any}]\!] = \mathcal{D}$$
$$[\![\texttt{Bottom}]\!] = \emptyset$$
$$T \leq S \Leftrightarrow [\![T]\!] \subseteq [\![S]\!]$$

where $\mathcal{D}$ represents the domain of all values.

Next we add data objects with structured tags. The tag of a value is accessed with `typeof(x)`. Each tag consists of a declared type name and some number of sub-expressions, written as $\texttt{Name}\{E_1, \cdots, E_n\}$. The center dots ($\cdots$) are meta-syntactic and represent a sequence of expressions. Tag types may have declared supertypes (written as `super(T)`). Any type used as a supertype must be declared as abstract, meaning it cannot have direct instances.

$$[\![\texttt{Name}\{\cdots\}]\!] = \{x \mid \texttt{typeof}(x) = \texttt{Name}\{\cdots\}\}$$
$$[\![\texttt{Abstract}\{\cdots\}]\!] = \bigcup_{\texttt{super}(T)=\texttt{Abstract}\{\cdots\}} [\![T]\!]$$

These types closely resemble the classes of an object-oriented language with generic (parametric) types, invariant type parameters, and no concrete inheritance. We prefer parametric *invariance* partly for reasons that have been addressed in the literature [12]. Invariance preserves the property that the only subtypes of a concrete type are `Bottom` and itself. This is important given how we map types to data representations: an `Array{Int}` cannot also be an `Array{Any}`, since those types imply different representations. If we tried to use covariance despite this, there would have to be some *other* notion of which type a value *really* had, which would be unsatisfyingly complex (tuples are a special case where covariance works, because each component type need only refer to single value, so there is no need for concrete tuple types with non-concrete parameters). We also find that most uses of covariance are more flexibly handled by union type connectives (introduced below).

Next we add conventional product (tuple) types, which are used to represent the arguments to methods. These are almost identical to the nominal types defined above, but are different in two ways: they are *covariant* in their parameters, and permit a special form ending in three dots (`...`) that denotes any number of trailing elements:

$$[\![\texttt{Tuple}\{P_1, \cdots, P_n\}]\!] = \prod_{1 \le i \le n} [\![P_i]\!]$$
$$[\![\texttt{Tuple}\{\cdots, P_n\ldots\}]\!], n \ge 1 = \bigcup_{i \ge n-1} [\![\texttt{Tuple}\{\cdots, P_n^i\}]\!]$$

P_n^i represents i repetitions of the final element P_n of the type expression.

The abstract tuple types ending in `...` correspond to variadic methods, which provide convenient interfaces for tasks like concatenating any number of arrays. Multiple dispatch has

been formulated as dispatch on tuple types before [22]. This formulation has the advantage that *any* type that is a subtype of a tuple type can be used to express the signature of a method. It also makes the system simpler and more reflective, since subtype queries can be used to ask questions about methods.

The types introduced so far would be perfectly sufficient for many programs, and are roughly equal in power to several multiple dispatch systems that have been designed before. However, these types are not closed under data-flow operations. For example, when the two branches of a conditional expression yield different types, a program analysis must compute the union of those types to derive the type of the conditional. The above types are not closed under set union. We therefore add the following type connective:

$$[\![\texttt{Union}\{A, B\}]\!] = [\![A]\!] \cup [\![B]\!]$$

As if by coincidence, `Union` types are also tremendously useful for expressing method dispatch. For example, if a certain method applies to all 32-bit integers regardless of whether they are signed or unsigned, it can be specialized for `Union{Int32,UInt32}`.

`Union` types are easy to understand, but complicate the type system considerably. To see this, notice that they provide an unlimited number of ways to rewrite any type. For example a type `T` can always be rewritten as `Union{T,Bottom}`, or `Union{Bottom,Union{T,Bottom}}`, etc. Any code that processes types must "understand" these equivalences. Covariant constructors (tuples in our case) also distribute over `Union` types, providing even more ways to rewrite types:

$$\texttt{Tuple}\{\texttt{Union}\{\texttt{A,B}\},\texttt{C}\} = \texttt{Union}\{\texttt{Tuple}\{\texttt{A,C}\},\texttt{Tuple}\{\texttt{B,C}\}\}$$

This is one of a few reasons that union types are often considered undesirable. When used with type inference, such types can grow without bound, possibly leading to slow or even non-terminating compilation. Their occurrence also typically corresponds to cases that would fail most static type checkers. Yet from the perspectives of both data-flow analysis

and method specialization, they are perfectly natural and even essential [19] [31] (TODO cite analyses that have used union types).

The next problem we need to solve arises from data-flow analysis of the `new` construct. When a type constructor `C` is applied to a type S that is known only approximately at compile time, the type `C`$\{S\}$ does not correctly represent the result. The correct result would be the union of all types `C`$\{T\}$ where $T \leq S$. Interestingly, there is again a corresponding need for such types in method dispatch. Often one has, for example, a method that applies to arrays of any kind of integer (`Array{Int32}`, `Array{Int64}`, etc.). These cases can be expressed using a `UnionAll` connective, which denotes an iterated union of a type expression for all values of a parameter in a specified range:

$$[\![\texttt{UnionAll } L\texttt{<:T<:}U\ A]\!] = \bigcup_{L \leq T \leq U} [\![A[T/\texttt{T}]]\!]$$

This is equivalent to an existential type [7]; for each concrete subtype of it there exists a corresponding T. Anecdotally, programmers often find existential types confusing. We prefer the union interpretation because we are describing sets of values; the notion of "there exists" can be semantically misleading since it sounds like only a single T value might be involved.

4.2.1 Examples

`UnionAll` types are quite expressive. In combination with nominal types they can describe groups of containers such as `UnionAll T<:Number Array{Array{T}}` (all arrays of arrays of some kind of number) or `Array{UnionAll T<:Number Array{T}}` (an array of arrays of potentially different types of number).

In combination with tuple types, `UnionAll` types provide powerful method dispatch specifications. For example `UnionAll T Tuple{Array{T},Int,T}` matches three arguments: an array, an integer, and a value that is an instance of the array's element type. This is a natural signature for a method that assigns a value to a given index within an array.

4.2.2 Type constructors

It is important for any proposed high-level technical computing language to be simple and approachable, since otherwise the value over established powerful-but-complex languages like C++ is less clear. In particular, type parameters raise usability concerns. Needing to write parameters along with every type is verbose, and requires users to know more about the type system and to know more details of particular types (how many parameters they have and what each one means). Furthermore, in many contexts type parameters are not directly relevant. For example, a large amount of code operates on `Array`s of any element type, and in these cases it should be possible to ignore type parameters.

Consider `Array{T}`, the type of arrays with element type `T`. In most languages with parametric types, the identifier `Array` would refer to a type constructor, i.e. a type of a different *kind* than ordinary types like `Int` or `Array{Int}`. Instead, we find it intuitive and appealing for `Array` to refer to any kind of array, so that a declaration such as `x::Array` simply asserts `x` to be some kind of array. In other words,

$$\texttt{Array} = \texttt{UnionAll T Array}'\texttt{\{T\}}$$

where $\texttt{Array}'$ refers to a hidden, internal type constructor. The { } syntax can then be used to instantiate a `UnionAll` type at a particular parameter value.

4.2.3 Associated types and type computation

Multiple dispatch systems have often only dispatched on the classes of arguments. This makes it appear necessary to introduce a separate template system to handle other kinds of parameterization.

4.2.4 Subtyping

Deciding subtyping for base types is straightforward: `Bottom` is a subtype of everything, everything is a subtype of `Any`, and tuple types are compared component-wise. The invariant

$$\frac{{}^{B}_{A}X^{L}, \Gamma \vdash T \leq S}{\Gamma \vdash \exists^{B}_{A}X.T \leq S} \qquad \frac{{}^{B}_{A}X^{R}, \Gamma \vdash T \leq S}{\Gamma \vdash T \leq \exists^{B}_{A}X.S}$$

$$\frac{{}^{B}X^{L}, \Gamma \vdash B \leq T}{{}^{B}X^{L}, \Gamma \vdash X \leq T} \qquad \frac{{}_{A}X^{L}, \Gamma \vdash T \leq A}{{}_{A}X^{L}, \Gamma \vdash T \leq X}$$

$$\frac{{}^{B}X^{L}, {}_{A}Y^{L}, \Gamma \vdash B \leq Y \ \vee \ X \leq A}{{}^{B}X^{L}, {}_{A}Y^{L}, \Gamma \vdash X \leq Y} \qquad \frac{{}^{B}_{A}X^{R}, Y^{R}, \Gamma \vdash B \leq A}{{}^{B}_{A}X^{R}, Y^{R}, \Gamma \vdash X \leq Y}$$

$$\frac{{}^{B}_{A}X^{R}, \Gamma \vdash A \leq T}{{}^{T}_{A}X^{R}, \Gamma \vdash X \leq T} \qquad \frac{{}^{B}_{A}X^{R}, \Gamma \vdash T \leq B}{{}^{B}_{A \cup T}X^{R}, \Gamma \vdash T \leq X}$$

Figure 4-1: Subtyping algorithm for `UnionAll` ($\exists$) and variables. X and Y are variables, A, B, T, and S are types. ${}^{B}_{A}X$ means X has lower bound A and upper bound B.

parameters of tag types are compared in both directions: to check `A{B}` $\leq$ `A{C}`, check `B` $\leq$ `C` and then `C` $\leq$ `B`. In fact, the algorithm depends on these checks being done in this order, as we will see in a moment.

Checking union types is a bit harder. When a union $A \cup B$ occurs in the algorithm, we need to nondeterministically replace it with either A or B. The rule is that for all such choices on the left of $\leq$, there must exist a set of choices on the right such that the rest of the algorithm answers "yes". This can be implemented by keeping a stack of decision points, and looping over all possibilities with an outer for-all loop and an inner there-exists loop. We speak of "decision points" instead of individual unions, since in a type like `Tuple{Union{A,B}...}` a single union might be compared many times.

The algorithm for `UnionAll` and variables is shown in figure 4-1. The first row says to handle a `UnionAll` by extending the environment with its variable, marked according to which side of $\leq$ it came from, and then recurring into the body. In analogy to union types, we need to check that for all variable values on the left, there exists a value on the right such

that the relation holds. The for-all side is relatively easy to implement, since we can just use a variable's bounds as proxies for its possible values (this is shown in the second row of the figure). We implement the there-exists side by narrowing a variable's bounds (raising the lower bound and lowering the upper bound). The relation holds as long as the bounds remain consistent (i.e. lower bound $\leq$ upper bound). The algorithm assumes that all input types are well-formed, which includes variable lower bounds always being less than or equal to upper bounds.

However, at this point (starting in the third row, second column) the algorithm appears asymmetric. This is a result of exploiting the lack of contravariant constructors. No contravariance means that every time a right-side variable appears on the *left* side of a comparison, it must be because it occurs in invariant position, and the steps outlined in the first paragraph of this section have "flipped" the order (comparing both $\texttt{B} \leq \texttt{C}$ and $\texttt{C} \leq \texttt{B}$). By symmetry, one would expect the bottom-left rule in the figure to update X's upper bound to $B \cap T$. But because of invariance, $T \leq B$ has already been checked by the bottom-right rule in the figure. Therefore $B \cap T = T$. This is the reason the "forward" direction of the comparison needs to be checked first: otherwise, we would have updated B to equal T already and the $T \leq B$ comparison would become vacuous. Alternatively, we could actually compute $B \cap T$. However there is reason to suspect that serious trouble lies that way. We would need either to add intersection types to the system, or compute a meet without them. Either way, the algorithm would become much more complex and, judging by past results, very likely undecidable.

TODO: termination and correctness.

These subtyping rules are very likely Π_2^{P}-hard. Checking a subtype relation with unions requires checking that for all choices on the left, there exists a choice on the right that makes the relation hold. This matches the quantifier structure of 2-TQBF problems of the form $\forall x_i.\exists y_i.F$ where F is a boolean formula. If the formula is rewritten in conjunctive normal form, it corresponds to subtype checking between two tuple types, where the relation must hold for each pair of corresponding types. Now use a type $\texttt{N}\{x\}$ to represent $\neg x$. The clause

$(x_i \vee y_i)$ can be translated to x_i `<: Union{N{`y_i`}, True}` (where the x_i and y_i are type variables bound by `UnionAll` on the left and right, respectively). We have not worked out the details, but this sketch is reasonably convincing. Π_2^{P} is only the most obvious reduction to try; it is possible our system equals PSPACE or even greater, as has often been the case for subtyping systems like ours.

4.2.5 Type system variants

features that are fairly straightforward to add:

- structurally-subtyped records
- mu-recursive types (regular trees)
- regular types (allowing ... in more places)

features that are difficult to add, or possibly break decidability:

- arrow types
- negations
- intersections, multiple inheritance
- universal quantifiers
- arbitrary predicates, theory of natural numbers, etc.

4.3 Dispatch mechanism

4.3.1 Ambiguities

4.4 Higher-order programming

Generic functions are first-class objects, and so can be passed as arguments just as in any dynamically-typed language with first-class functions. However, assigning useful type tags

to generic functions and deciding how they should dispatch is not so simple. Past work has often described the types of generic functions using the "intersection of arrows" formalism [30] [14] [7] [8]. Since an ordinary function has an arrow type $A \to B$ describing how it maps arguments A to results B, a function with multiple definitions can natually be considered to have multiple such types. For example, a `sin` function with the following two definitions:

```
sin(x::Float64) = # compute sine of x in double precision
sin(v::Vector) = map(sin, v)
```

could have the type $(\texttt{Float64} \to \texttt{Float64}) \cap (\texttt{Vector} \to \texttt{Vector})$. The intuition is that this `sin` function can be used both where a $\texttt{Float64} \to \texttt{Float64}$ function is expected and where a $\texttt{Vector} \to \texttt{Vector}$ function is expected, and therefore its type is the intersection of these types.

This approach is effective for statically checking uses of generic functions: anywhere a function goes, we must keep track of which arrow types it "contains" in order to be sure that at least one matches every call site and allows the surrounding code to type check. However, despite the naturalness of this typing of generic functions, this formulation is quite problematic for dispatch and code specialization (not to mention that it might make subtyping undecidable).

4.4.1 Problems for code selection

Consider what happens when we try to define an integration function:

```
# 1-d integration of a real-valued function
integrate(f::Float64->Float64, x0, x1)

# multi-dimensional integration of a vector-valued function
integrate(f::Vector->Vector, v0, v1)
```

The `->` is not real Julia syntax, but is assumed for the sake of this example. Here we wish to select a different integration routine based on what kind of function is to be integrated. However, these definitions are ambiguous with respect to the `sin` function defined above. Of course, the potential for method ambiguities existed already. However this sort of ambiguity

is introduced *non-locally* — it cannot be detected when the `integrate` methods are defined.

Such a non-local introduction of ambiguity is a special case of the general problem that a generic function's type would change depending on what definitions have been added (which depends e.g. on which libraries have been loaded). This does not feel like the right abstraction: type tags are supposed to form a "ground truth" about objects against which program behavior can be selected. Though generic functions change with the addition of methods, it would be more satisfying for their types to somehow reflect an intrinsic, unchanging property.

An additional minor problem with the intersection of arrows interpretation is that we have found, in practice, that Julia methods often have a large number of definitions. For example, the `+` function in Julia v0.3.4 has 117 definitions, and in a more recent development version with more functionality, it has 150 methods. An intersection of 150 types would be unwieldy, even if only used when debugging the compiler.

A slightly different approach we might try would be to immitate the types of higher-order functions in traditional statically-typed functional languages. For example, we might wish to write `map` as follows:

```
map{A,B}(f::A->B, x::List{A}) =
  isempty(x) ? List{B}() : List{B}(f(head(x)), map(f, tail(x)))
```

The idea is for the first argument to match any function, and not use the arrow type for dispatch, thereby avoiding ambiguity problems. Instead, immediately after method selection, values for `A` and `B` would be determined using the element type of `x` and the table of definitions of `f`.

Unfortunately it is not clear how exactly `B` should be determined. We could require return type declarations on every method, but this would adversely affect usability (such declarations would also be helpful if we wanted to dispatch on arrow types, though they would not solve the ambiguity problem). Or we could use type inference of `f` on argument type `A`. This would not work very well, since the result would depend on partly-arbitrary heuristics. Such heuristics are fine for analyzing a program, but are not appropriate for determining the

value of a user-visible variable, as this would make program behavior unpredictable.

4.4.2 Problems for code specialization

For code specialization to be effective, it must eliminate as many irrelevant cases as possible. Intersection types seem to be naturally opposed to this process, since they have the ability to generate infinite descending chains of ever-more-specific function types by tacking on more terms with $\cap$. There would be no such thing as a maximally-specific function type. In particular, there would be no good way to express that a function has exactly one definition, which is an especially important case for optimizing code.

For example, say we have a definition `f(g::String->String)`, and a function `h` with a single `Int` $\rightarrow$ `Int` definition. Naturally, `f` is not applicable to `h`. However, given the call site `f(h)`, we are forced to conclude that `f` might be called with a function of type (`Int` $\rightarrow$ `Int`) $\cap$ (`String` $\rightarrow$ `String`), since in general `Int` $\rightarrow$ `Int` might be only an approximation of the true type of the argument.

The other major concern when specializing code is whether, having generated code for a certain type, we would be able to reuse that code often enough for the effort to be worthwhile. In the case of arrow types, this equates to asking how often generic functions share the same set of signatures. This question can be answered empirically. Studying the Julia `Base` library as of this writing, there are 1059 generic functions. We examined all 560211 pairs of functions; summary statistics are shown in figure 4-2. Overall, it is rare for functions to share type signatures. Many of the 85 functions with matches (meaning there exists some other function with the same type) are predicates, which all have types similar to `Any` $\rightarrow$ `Bool`. The mean of 0.23 means that if we pick a function uniformly at random, on average 0.23 other functions will match it. The return types compared here depend on our heuristic type inference algorithm, so it useful to exclude them in order to get an upper bound. If we do that, and only consider arguments, the mean number of matches rises to 1.73.

The specific example of the `sin` and `cos` functions provides some intuition for why there are so few matches. One would guess that the type behavior of these functions would be

	matching pairs	GFs with matches	mean
arguments only	1831 (0.327%)	329	1.73
arguments and return types	241 (0.043%)	85	0.23

Figure 4-2: Number and percentage of pairs of functions with matching arguments, or matching arguments and return types. The second column gives the number of functions that have matches. The third column gives the mean number of matches per function.

identical, however the above evaluation showed this not to be the case. The reason is that the functions have definitions to make them operate elementwise on both dense and sparse arrays. `sin` maps zero to zero, but `cos` maps zero to one, so `sin` of a sparse array gives a sparse array, but `cos` of a sparse array gives a dense array.

4.4.3 Possible solutions

The general lack of sharing of generic function types suggests the first possible solution: give each generic function a new type that is uniquely associated with it. For example, the type of `sin` would be `GenericFunction{sin}`. This type merely identifies the function in question, and says nothing more about its behavior. It is easy to read, and easily specific enough to avoid ambiguity and specialization problems. It does *not* immediately solve the above problem of determining the result type of `map`. However there are corresponding performance benefits, since specialzing code for a specific function argument naturally lends itself to inlining.

Another approach that is especially relevant to technical computing is to use nominal function types. In mathematics, the argument and return types of a function are often among its least interesting properties. In some domains, for example, all functions can implicitly be assumed $\mathbb{R} \to \mathbb{R}$, and the interesting property might be what order of integrable singularity is present (see section 5.3 for an application), or what dimension of linear operator the function represents. The idea of nominal function types is to describe the properties of interest using a data object, and then allow that data object to be treated as a function, i.e. "called".

Some object-oriented languages call such an object a *functor*.

Julia accomodates this approach with a small adjustment to the evaluation rules: in the application syntax $e_1(e_2)$ when e_1 is not a function, evaluate `call`(e_1, e_2) instead, where `call` is a particular generic function known to the system. Now we can define

```
immutable Arrow{A,B}
    f
end

call{A,B}(a::Arrow{A,B}, x::A) = a.f(x)::B
```

4.4.4 Implementing `map`

4.5 Performance model

4.5.1 Type inference

4.5.2 Specialization

Chapter 5

Case studies

5.1 Explication through elimination

This section will illustrate how we implement key features of technical computing systems using our methodology.

5.1.1 Conversion and comparison

Type conversion provides a classic example of a binary method. Multiple dispatch allows us to avoid deciding whether the converted-to type or the converted-from type is responsible for defining the operation. Defining a specific conversion is straightforward, and might look like this in Julia syntax:

```
convert(::Type{Float64}, x::Int32) = ...
```

A call to this method would look like `convert(Float64, 1)`.

Using conversion in generic code requires more sophisticated definitions. For example we might need to convert one value to the type of another, by writing `convert(typeof(y), x)`. What set of definitions must exist to make that call work in all reasonable cases? Clearly we cannot explicitly write all $O(n^2)$ possibilities. We need abstract definitions that cover many points in the dispatch matrix. One such family of points is particularly important: those

that describe converting a value to a type it is already an instance of. In our system this can be handled by a single definition that performs "triangular" dispatch:

```
convert{T,S<:T}(::Type{T}, x::S) = x
```

"Triangular" refers to the rough shape of the dispatch matrix covered by such a definition: for all types `T` in the first argument slot, match values of any type less than it in the second argument slot.

A similar trick is useful for defining equivalence relations. It is most likely unavoidable for programming languages to need multiple notions of equality. Two in particular are natural: an *intensional* notion that equates objects that look identical, and an *extensional* notion that equates objects that mean the same thing for some standard set of purposes. Intensional equality (`===` in Julia, mentioned in section 4.1) lends itself to being implemented once by the language implementer, since it can work by directly comparing the representations of objects. Extensional equality (`==` in Julia), on the other hand, must be extensible to user-defined data types. The latter function must call the former in order to have any basis for its comparisons.

As with conversion, we would like to provide default definitions for `==` that cover families of cases. Numbers are a reasonable domain to pick, since all numbers should be equality-comparable to each other. We might try

```
==(x::Number, y::Number) = x === y
```

meaning that number comparison can simply fall back to intensional equality. However this definition is rather dubious. It gets the wrong answer every time the arguments are different representations (e.g. integer and floating point) of the same quantity. We might hope that its behavior will be "patched up" later by more specific definitions for various concrete number types, but it still covers a dangerous amount of dispatch space. If later definitions somehow miss a particular combination of number types, we could get a silent wrong answer instead of an error. (Note that statically checking method exhaustiveness is no help here.)

"Diagonal" dispatch lets us improve the definition:

```
==={T<:Number}(x::T, y::T) = x === y
```

Now `===` will only be used on arguments of the same type, making it far more likely to give the right answer. Even better, any case where it does not give the right answer can be fixed with a single definition, i.e. `==(x::S, y::S)` for some concrete type `S`. The more general `(Number, Number)` case is left open, and in the next section we will take advantage of this to implement "automatic" type promotion.

5.1.2 Numeric types and embeddings

We might prefer "number" to be a single, concrete concept, but the history of mathematics has seen the concept extended many times, from integers to rationals to reals, and then to complex, quaternion, and more. These constructions tend to follow a pattern: a new set of numbers is constructed around a subset isomorphic to an existing set of numbers. For example, the reals are isomorphic to the complex numbers with zero imaginary part.

Human beings happen to be good at equating and moving between isomorphic sets, so it is easy to imagine that the reals and complexes with zero imaginary part are one and the same. But a computer forces us to be specific, and admit that a real number is not complex, and a complex number is not real. And yet the close relationship between them is too compelling not to model in a computer somehow. Here we have a numerical analog to the famous "circle and ellipse" problem in object-oriented programming: the set of circles is isomorphic to the set of ellipses with equal axes, yet neither "is a" relationship in a class hierarchy seems fully correct. An ellipse is not a circle, and in general a circle cannot serve as an ellipse (for example, the set of circles is not closed under the same operations that the set of ellipses is, so a program written for ellipses might not work on circles). This problem implies that a single built-in type hierarchy is not sufficient: we want to model custom *kinds* of relationships between types (e.g. "can be embedded in" in addition to "is a").

Current approaches

Numbers tend to be among the most complex features of a language. Numeric types usually need to be a special case: in a typical language with built-in numeric types, describing their behavior is beyond the expressive power of the language itself. For example, in C arithmetic operators like `+` accept multiple types of arguments (ints and floats), but no user-defined C function can do this (this situation is of course improved in C++). In Python, a special arrangement is made for `+` to call either an `__add__` or `__radd__` method, effectively providing double-dispatch for arithmetic in a language that is idiomatically single-dispatch.

Implementing type embeddings

Most functions are naturally implemented in the value domain, but some are actually easier to implement in the type domain. One reason is that there is a bottom element, which most data types lack.

It has been suggested on theoretical grounds [29] that generic binary operators should have "key variants" where the arguments are of the same type.

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

+{T<:Number}(x::T, y::T) = no_op_err("+", T)
```

A full-featured promotion operator is a tall order. We would like

- each combination of types only needs to be defined in one order - it falls back to join - must prevent promotion above a certain point to avoid circularity in promoting fallback definitions of operators.

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

promote_type{T,S}(::Type{T}, ::Type{S}) =
    promote_result(T, S, promote_rule(T,S), promote_rule(S,T))

promote_rule(T, S) = Bottom

promote_result(t,s,T,S) = promote_type(T,S)
```

```
# If no promote_rule is defined, both directions give Bottom. In that
# case use typejoin on the original types instead.
promote_result{T,S}(::Type{T},::Type{S},::Type{Bottom},::Type{Bottom}) = typejo


# Because of the promoting fallback definitions for Number, we need
# a special case for undefined promote_rule on numeric types.
# Otherwise, typejoin(T,S) is called (returning Number) so no conversion
# happens, and +(promote(x,y)...) is called again, causing a stack
# overflow.
promote_result{T<:Number,S<:Number}(::Type{T},::Type{S},::Type{Bottom},::Type{B
    promote_to_super(T, S, typejoin(T,S))

# promote numeric types T and S to typejoin(T,S) if T<:S or S<:T
# for example this makes promote_type(Integer,Real) == Real without
# promoting arbitrary pairs of numeric types to Number.
promote_to_super{T<:Number          }(::Type{T}, ::Type{T}, ::Type{T}) = T
promote_to_super{T<:Number,S<:Number}(::Type{T}, ::Type{S}, ::Type{T}) = T
promote_to_super{T<:Number,S<:Number}(::Type{T}, ::Type{S}, ::Type{S}) = S
promote_to_super{T<:Number,S<:Number}(::Type{T}, ::Type{S}, ::Type) =
    error("no promotion exists for ", T, " and ", S)
```

Application to ranges

Ranges illustrate an interesting application of type promotion. A range data type, notated `a:s:b`, represents a sequence of values starting at `a` and ending at `b`, with a distance of `s` between elements (internally, this notation is translated to `colon(a, s, b)`). Ranges seem simple enough, but a reliable, efficient, and generic implementation is difficult to achieve. We propose the following requirements:

- The start and stop values can be passed as different types, but internally should be of the same type.
- Ranges should work with ordinal types, not just numbers (examples include characters, pointers, and calendar dates).
- If any of the arguments is a floating-point number, a special `FloatRange` type designed to cope well with roundoff is returned.

In the case of ordinal types, the step value is naturally of a different type than the elements of the range. For example, one may add 1 to a character to get the "next" encoded character, but it does not make sense to add two characters.

It turns out that the desired behavior can be achieved with six definitions:

First, given three floats of the same type we can construct a `FloatRange` right away:

```
colon{T<:FloatingPoint}(start::T, step::T, stop::T) = FloatRange{T}(start, step, stop)
```

Next, if `a` and `b` are of the same type and there are no floats, we can construct a general range:

```
colon{T}(start::T, step, stop::T) = StepRange(start, step, stop)
```

Now there is a problem to fix: if the first and last arguments are of some non-floating-point numeric type, but the step is floating point, we want to promote all arguments to a common floating point type. We must also do this if the first and last arguments are floats, but the step is some other kind of number:

```
colon{T<:Real}(a::T, s::FloatingPoint, b::T) = colon(promote(a,s,b)...)
```

```
colon{T<:FloatingPoint}(a::T, s::Real, b::T) = colon(promote(a,s,b)...)
```

These two definitions are correct, but ambiguous: if the step is a float of a different type than `a` and `b` both definitions are equally applicable. We can add the following disambiguating definition:

```
colon{T<:FloatingPoint}(a::T, s::FloatingPoint, b::T) = colon(promote(a,s,b)...)
```

All of these five definitions require `a` and `b` to be of the same type. If they are not, we must promote just those two arguments, and leave the step alone (in case we are dealing with ordinal types):

```
colon{A,B}(a::A, s, b::B) = colon(convert(promote_type(A,B),a), s, convert(promote_type(
```

This example shows that it is not always sufficient to have a built-in set of "promoting operators". Library functions like this `colon` need more control.

Diversity of number and number-like types in practice

Originally, our reasons for implementing all numeric types at the library level were not entirely practical. We had a principled opposition to including such definitions in a compiler, and guessed that being able to define numeric types would help ensure the language was "powerful enough". However, defining numeric and number-like types and their interactions turns out to be surprisingly useful. Once such types are easy to obtain, people find more and more uses for them.

Even among basic types that might reasonably be "built in", there is enough complexity to require an organizational strategy. We might have

- Ordinal types `Pointer`, `Char`
- Integer types `Bool`, `Int8`, `Int16`, `Int32`, `Int64`, `Int128`, `UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `BigInt`
- Floating point types `Float16`, `Float32`, `Float64`, `Float80`*, `Float128`*, `BigFloat`, `DoubleDouble`
- Extensions `Rational`, `Complex`, `Quaternion`

Types with * have not been implemented yet, but the rest have. In external packages, people have implemented types for interval arithmetic, dual and hyper-dual numbers for computing first and second derivatives, and finite fields.

Some applications benefit in performance from fixed-point arithmetic. This has been implemented in a package as `Fixed32{b}`, where the number of fraction bits is a parameter.

A problem in the design of image processing libraries was solved by defining a new kind of fixed-point type [11]. The problem is that image scientists often want to work with fractional pixel values in the interval $[0, 1]$, but most graphics libraries (and memory efficiency concerns) require 8-bit integer pixel components with values in the interval $[0, 255]$. The solution is a `Ufixed8` type that uses an unsigned 8-bit integer as its representation, but behaves like a fraction over 255.

Many real-world quantities are not numbers exactly, but benefit from the same mechanisms in their implementation. Examples include colors (which form a vector space, and where many different useful bases have been standardized), physical units, and DNA nucleotides. Date and time arithmetic is especially intricate and irregular (more on this in section 5.4).

5.1.3 Multidimensional array indexing

One-dimensional arrays are a simple and essential data structure found in most programming languages. The multi-dimensional arrays required in scientific computing, however, are a different beast entirely. Allowing any number of dimensions entails a significant increase in complexity. Why? The essential reason is that core properties of the data structure no longer fit in a constant amount of space. The space needed to store the sizes of the dimensions (the array shape) is proportional to the number of dimensions. This does not seem so bad, but becomes a large problem due to three additional facts.

First, code that operates on the dimension sizes needs to be highly efficient. Typically the overhead of a loop is unacceptable, and such code needs to be fully unrolled. Second, in some code the number of dimensions is a *dynamic* property — it is only known at run time. Third, programs may wish to treat arrays with different numbers of dimensions very differently. A vector (1d) might have rather different behaviors than a matrix (2d) (for example, to compute a norm). This kind of behavior makes the number of dimensions a crucial part of program semantics, preventing it from remaining a compiler implementation detail.

These facts pull in different directions. The first fact asks for static analysis. The second fact asks for run-time flexibility. The third fact asks for dimensionality to be part of the type system, but partly determined at run time (for example, via virtual method dispatch). Current approaches choose a compromise. In some systems, the number of dimensions has a strict limit (e.g. 3 or 4), so that separate classes for each case may be written out in full. Other systems choose flexibility, and just accept that most or all operations will be

dynamically dispatched. Other systems might provide flexibility only at compile time, for example a template library where the number of dimensions must be statically known.

Whatever decision is made, rules must be defined for how various operators act on dimensions. For now we will focus on indexing, since selecting parts of arrays has particularly rich behavior with respect to dimensionality. For example, if a single row or column of a matrix is selected, does the result have one or two dimensions? Array implementations prefer to invoke general rules to answer such questions. Such a rule might say "dimensions indexed with scalars are dropped", or "trailing dimensions of size one are dropped", or "the rank of the result is the sum of the ranks of the indexes" (as in APL).

Our goal here is a bit unusual: we are not concerned with which rules might work best, but merely with how they can be specified, so that domain experts can experiment. In fact different domains want different things. Working with images, each dimension might be quite different, e.g. representing time, space, or color, so you don't want to drop or rearrange dimensions very often.

How are such rules implemented? For a language with built-in multidimensional arrays, the compiler will analyze indexing expressions and determine an answer using hard-coded logic. However, this approach is not satisfying: we would rather implement the behavior in libraries, so that different kinds of arrays may be defined, or so that rules of similar complexity may be defined for other kinds of objects. But these kinds of rules are unusually difficult to implement in libraries. If a library writes out its indexing logic using imperative code, the host language compiler is not likely to be able to analyze it. Using compile-time abstracton (templates) would provide better performance, but such libraries tend to be difficult to write (and read), and the full complement of indexing behavior expected by technical users strains the capabilities of such systems.

Our dispatch mechanism permits a novel solution. If a multiple dispatch system supports variadic functions and argument "splicing" (the ability to pass a structure of n values as n separate arguments to a function), then indexing behavior can be defined as method signatures.

This solution is still a compromise among the factors outlined above, but it is a new compromise that provides a modest, but definite, increment of power.

Below we define a function `index_shape` that computes the shape of a result array given a series of index arguments. We show three versions, each implementing a different rule that users in different domains might want:

```
# drop dimensions indexed with scalars
index_shape() = ()
index_shape(i::Real, I...) = index_shape(I...)
index_shape(i, I...) = tuple(length(i), index_shape(I...)...)

# drop trailing dimensions indexed with scalars
index_shape(i::Real...) = ()
index_shape(i, I...) = tuple(length(i), index_shape(I...)...)

# rank summing (APL)
index_shape() = ()
index_shape(i, I...) = tuple(size(i)..., index_shape(I...)...)
```

Inferring the length of the result of `index_shape` is sufficient to infer the rank of the result array.

These definitions are concise, easy to write, and possible for a compiler to understand fully using straightforward techniques.

The result type is determined using only dataflow type inference, plus a rule for splicing an immediate container (the type of `f((a,b)...)` is the type of `f(a,b)`). Argument list destructuring takes place inside the type intersection operator used to combine argument types with method signatures.

This approach does not depend on any heuristics. Each call to `index_shape` simply requires one recursive invocation of type inference. This process reaches the base case `()` for these definitions, since each recursive call handles a shorter argument list (for less-well-behaved definitions, we might end up invoking a widening operator instead).

5.1.4 Array views

tim holy in issue 8839:

"without staged functions in my initial post in 8235. The take-home message: generating all methods through dimension 8 resulted in more than 5000 separate methods, and required over 4 minutes of parsing & lowering time (i.e., a 4-minute delay while compiling julia). By comparison, the stagedfunction implementation loads in a snap, and of course can go even beyond 8 dimensions."

5.1.5 Units

5.1.6 Even more elimination?

Some features of the language could be even further eliminated. For example data types could be implemented in terms of lambda abstractions. But certain patterns are so useful that they might as well be provided in a standard form. It also probably makes the compiler much more efficient not to need to pass around and repeatedly analyze full representations of the meanings of such ubiquitous constructs.

5.2 Numerical linear algebra

The crucial roles of code selection and code specialization in technical computing are captured well in the linear algebra software engineer's dilemma, described in the context of LAPACK and ScaLAPACK by Demmel and Dongarra, et.al. [13]:

```
(1) for all linear algebra problems (linear systems, eigenproblems, ...)
(2)    for all matrix types (general, symmetric, banded, ...)
(3)       for all data types (real, complex, single, double, higher precision)
(4)          for all machine architectures
                and communication topologies
(5)                for all programming interfaces
(6)                   provide the best algorithm(s) available in terms of
                      performance and accuracy (``algorithms" is plural
                      because sometimes no single one is always best)
```

Multiple dispatch on special matrices

29 LAPACK types via composition of 9 types, issue 8240

QR factorization: want to support it on all input types, but many types (integer, rational) are not closed under the needed operations. compare to eigen: in C++, if you want a qrfact of an integer matrix, you might get a compile-time error, or it might work but all the values will be truncated when stored. this is a big generic programming challenge. you can't expect types to have a "type to use for QR fact" trait.

qrfact computes the type of the result needed:

```
function qrfact{T}(A::StridedMatrix{T}; pivot=false)
    S = typeof(one(T)/norm(one(T)))
    if S != T
        qrfact!(convert(AbstractMatrix{S},A), pivot=pivot)
    else
        qrfact!(copy(A),pivot=pivot))
    end
end
```

This code is not statically typeable, and yet with specialization a compiler could in fact determine the type of each call site. It just happens to be convenient to specify this behavior with a branch.

5.3 Boundary-element method

There are lots of general packages for FEM problems, but it is much more difficult to create such a package for BEM problems. The method requires integrating functions with singularities, many times in the inner loop of code that builds the problem matrix. Integrating such functions numerically on each iteration is much too slow. As a result, many special-purpose implementations have been written by hand for different problems.

Some recent work (TODO cite Homer) managed a more general solution, using Mathematica to generate C++ code for different cases. This worked well, but was difficult to implement and the resulting system is difficult to use. We see the familiar pattern of using multiple languages and code-generation techniques, with coordination of the overall process

done either manually or with ad-hoc scripts. To polish the implementation for use as a practical library, a likely next step would be to add a Python interface, adding yet another layer of complexity.

The code can be structured as a simple function library.

5.3.1 Galerkin matrix assembly for singular kernels

A typical problem in computational science is to form a discrete approximation of some infinite-dimensional linear operator $\mathcal{L}$ with some finite set of basis functions $\{b_m\}$ via a Galerkin approach [refs], which leads to a matrix L with entries $L_{mn} = \langle b_m, \mathcal{L}b_n \rangle = \langle b_m, b_n \rangle_{\mathcal{L}}$ where $\langle \cdot, \cdot \rangle$ denotes some inner product (e.g. $\langle u, v \rangle = \int uv$ is typical) and $\langle \cdot, \cdot \rangle_{\mathcal{L}}$ is the *bilinear form* of the problem. Computing these matrix elements is known as the matrix *assembly* step, and its performance is a crucial concern for solving partial differential equations (PDEs) and integral equations (IEs).

The "easy" case: nonsingular assembly

For example, in the finite-element method (FEM) [refs], the basis functions b_m are typically low-order polynomials defined piecewise over geometric elements (typically triangles or tetrahedra), and $\mathcal{L}$ is typically a differential operator like $-\nabla \cdot c(x)\nabla$ for some coefficients $c(x)$, which leads to a bilinear form $\langle b_m, b_n \rangle_{\mathcal{L}} = \int \nabla b_m \cdot c(x) \nabla b_n$ (after integration by parts). Because the basis functions are localized and $\mathcal{L}$ consists of local operations, the matrix L is sparse and L_{mn} need only be computed for m and n corresponding to neighboring elements. Moreover, these integrals are straightforward to evaluate by standard cubature schemes because the integrands are *nonsingular*: they typically have no divergences or discontinuities. In particular, because the functions b_m and c are usually smooth within a single element, one can use a fixed low-order cubature rule: you evaluate the integrand at a handful of precomputed points within each element, multiply by precomputed weights, and sum to obtain the approximate integral.

Even so, the basis functions and the coefficient function $c(x)$ may need to be evaluated

tens of millions of times for even a moderate-size mesh in three dimensions, so production FEM implementations in traditional high-level dynamic languages such as Matlab and Python are forced to offload matrix assembly to external C and C++ code. For example, the popular FEniCS [ref] and Firedrake [ref] FEM packages for Python both implement domain-specific compilers: a symbolic expression for the bilinear form is combined with fragments of user-specified C++ code to define functions like $c(x)$, compiled to C++ code, and then compiled to object code which is dynamically loaded. In Julia, we believe this could be simplified considerably because functions like $c(x)$ could be defined directly in Julia and code generation/compilation could be performed entirely within Julia without a C++ intermediary. Indeed, preliminary experiments with pure Julia FEM implementations [ref http://www.codeproject.com/Articles/579983/Finite-Element-programming-in-Julia] have demonstrated performance comparable to sophisticated solutions like FEniCS in Python and FreeFem++ in C++ [ref]

Singular assembly for integral operators

A much more challenging case of Galerkin matrix assembly arises for singular *integral* operators $\mathcal{L}$, which act by convolving their operand against a singular "kernel" function $K(x)$: $u = \mathcal{L}v$ means that $u(x) = \int K(x-x')v(x')dx'$. For example, in electrostatics and other Poisson problems, the kernel is $K(x) = 1/|x|$ in three dimensions and $\ln|x|$ in two dimensions, while in scalar Helmholtz (wave) problems it is $e^{ik|x|}/|x|$ in three dimensions and a Hankel function $H_0^{(1)}(k|x|)$ in two dimensions. Formally, Galerkin discretizations lead to matrix assembly problems similar to those above: $L_{mn} =: \langle b_m, \mathcal{L}b_n\rangle = \int b_m(x)K(x-x')b_n(x')dx\,dx'$. However, there are several important differences from FEM:

- The kernel $K(x)$ nearly always diverges for $|x| = 0$, which means that generic cubature schemes are either unacceptably inaccurate (for low-order schemes) or unacceptably costly (for adaptive high-order schemes, which require huge numbers of cubature points around the singularity), or both.

- Integral operators typically arise for *surface* integral equations (SIEs) [ref], and involve unknowns on a surface. The analogue of the FEM discretization is then a boundary element method (BEM) [ref], which discretizes a surface into elements (e.g. triangles), with basis functions that are low-order polynomials defiend piecewise in the elements. However, there are also volume integral equations (VIEs) which have FEM-like volumetric meshes and basis functions.

- The matrix L is typically dense, since K is long-range. For large problems, L is often stored and applied implicitly via fast-multipole methods [refs] and similar schemes, but even in this case the diagonal L_{mm} and the entries L_{mn} for adjacent elements must typically be computed explicitly. (Moreover, these are the integrals in which the K singularity is present.)

These difficulties are part of the reason why there is currently *no* truly "generic" BEM software, analogous to FEniCS for FEM: essentially all practical BEM code is written for a specific integral kernel and a specific class of basis functions arising in a particular physical problem. Changing anything about the kernel or the basis—for example, going from two- to three-dimensional problems—is a major undertaking.

We believe that Julia should be an ideal platform on which to attack this problem:

- Multiple dispatch allows the cubature scheme to be selected at compile-time based on the dimensionality, the degree of the singularity, the degree of the polynomial basis, and so on, and allows specialized schemes to be added easily for particular problems with no runtime penalty.

- Staged functions allow computer-algebra systems to be invoked at compile-time to generate specialized cubature schemes for particular kernels. New developments in BEM integration schemes [ref Homer] have provided efficient cubature-generation algorithms of this sort, but it has not yet been practical to integrate them with runtime code in a completely automated way.

A prototype implementation of this approach follows.

5.4 Dates

compare to python DateTime, compare code length

5.5 JUMP

Metaprogramming tools reused for symbolic algebra

5.6 Computational geometry

robust predicates (dispatch over points, lines) and VoronoiDelaunay.jl benchmarked against CGAL

5.7 Beating the incumbents

- erfinv and digamma using horner macro

- randn beating matlab

`@evalpoly` macro has separate cases for real and complex in order to automatically take advantage of a subtle algorithm (TODO cite TAOCP). A macro is perfect for generating the neccessary code, however it lacks the type information needed to select between the two cases. In Julia, it can generate a branch with a type check, and rely on the unused case being removed by automatic specialization. (A generic function with two definitions could be generated instead, but in high-performance programming the "force inlining" behavior of macros is welcome.)

If nothing else, demonstrates that removing glue code overhead is worthwhile.

grisu: 6kLOC to 1kLOC (PR 7291)

Chapter 6

Conclusion

A generation of dynamic languages have been designed by trying variants of the class-based object oriented paradigm. This process has been aided by the development of standard techniques (e.g. bytecode VMs) and reusable infrastructure such as code generators, garbage collectors, and whole VMs like the JVM and CLR.

It is possible to envision a future generation of languages that generalize this design to set-theoretic subtyping instead of just classes. This next generation will require its own new tools, such as partial evaluators (already under development in PyPy and Truffle). One can also imagine these future language designers wanting reusable program analyses, and tools for developing lattices and their operators.

It is interesting to observe that the data model of a language like Julia consists of two key relations: the subtype relation, which is relatively well understood and enjoys useful properties like transitivity, but also the typeof relation, which relates individual values to their types (i.e. the ‘typeof‘ function). The typeof relation appears not to be transitive, and also has a degree of arbitrariness: a value is of a type merely because it is labeled as such, and because various bits of code conspire to ensure that this labeling makes sense according to various criteria.

We have speculated about whether future languages will be able to do away with this distinction. One approach is $\lambda_{\aleph}$. We have also speculated that this could be done using types

based on non-well-founded set theory, combining the subset-of and element-of relations using self-containing sets. We are not yet sure what a practical language based on this idea might look like.

There are several key aspects of performance programming that our design does not directly address.

Talk about storage and in-place optimizations.

Appendix A

Subtyping algorithm

```
abstract Ty

type TypeName
  super::Ty
  TypeName() = new()
end

type TagT <: Ty
  name::TypeName
  params
  vararg::Bool
end

type UnionT <: Ty
  a; b
end

type Var
  lb; ub
end

type UnionAllT <: Ty
  var::Var
  T
end

## Any, Bottom, and Tuple
const AnyT = TagT(TypeName(), ());  AnyT.name.super = AnyT
type BottomTy <: Ty; end; const BottomT = BottomTy()
const TupleName = TypeName(); TupleName.super = AnyT

## type application
inst(t::TagT) = t
inst(t::UnionAllT, param) = subst(t.T, Dict{Any,Any}(t.var => param))
inst(t::UnionAllT, param, rest...) = inst(inst(t,param), rest...)
super(t::TagT) = inst(t.name.super, t.params...)

extend(d::Dict, k, v) = (x = copy(d); x[k]=v; x)
subst(t,          env) = t
subst(t::TagT,    env) =
```

```
  t===AnyT ? t : TagT(t.name, map(x->subst(x,env), t.params), t.vararg)
subst(t::UnionT,  env) = UnionT(subst(t.a,env), subst(t.b,env))
subst(t::Var,     env) = get(env, t, t)
function subst(t::UnionAllT, env)
  newVar = Var(subst(t.var.lb,env), subst(t.var.ub,env))
  UnionAllT(newVar, subst(t.T, extend(env, t.var, newVar)))
end

rename(t::UnionAllT) = let v = Var(t.var.lb, t.var.ub)
  UnionAllT(v, inst(t,v))
end

type Bounds
  lb; ub       # current lower and upper bounds of a Var
  right::Bool  # this Var is on the right-hand side of A <: B
end

type UnionState
  depth::Int           # number of union decision points we're inside
  more::Bool           # new union found; need to grow stack
  stack::Vector{Bool}  # stack of decisions
  UnionState() = new(1,0,Bool[])
end

type Env
  vars::Dict{Var,Bounds}
  Lunions::UnionState
  Runions::UnionState
  Env() = new(Dict{Var,Bounds}(), UnionState(), UnionState())
end

issub(x, y)              = forall_exists_issub(x, y, Env(), false)
issub(x, y, env)         = (x === y)
issub(x::Ty, y::Ty, env) = (x === y) || x === BottomT

function forall_exists_issub(x, y, env, anyunions::Bool)
  for forall in false:anyunions
    if !isempty(env.Lunions.stack)
      env.Lunions.stack[end] = forall
    end

    !exists_issub(x, y, env, false) && return false

    if env.Lunions.more
      push!(env.Lunions.stack, false)
```

```
      sub = forall_exists_issub(x, y, env, true)
      pop!(env.Lunions.stack)
      !sub && return false
    end end
  return true
end

function exists_issub(x, y, env, anyunions::Bool)
  for exists in false:anyunions
    if !isempty(env.Runions.stack)
      env.Runions.stack[end] = exists
    end
    env.Lunions.depth = env.Runions.depth = 1
    env.Lunions.more = env.Runions.more = false

    found = issub(x, y, env)

    if env.Lunions.more
      return true  # return up to forall_exists_issub
    end
    if env.Runions.more
      push!(env.Runions.stack, false)
      found = exists_issub(x, y, env, true)
      pop!(env.Runions.stack)
    end
    found && return true
  end
  return false
end

function issub_union(t, u::UnionT, env, R, state::UnionState)
  if state.depth > length(state.stack)
    state.more = true
    return true
  end
  ui = state.stack[state.depth]; state.depth += 1
  choice = u.(1+ui)
  return R ? issub(t, choice, env) : issub(choice, t, env)
end

issub(a::UnionT, b::UnionT, env) = issub_union(a, b, env, true, env.Runions)
issub(a::UnionT, b::Ty, env)     = issub_union(b, a, env, false, env.Lunions)
issub(a::Ty, b::UnionT, env)     = issub_union(a, b, env, true, env.Runions)
# take apart unions before handling vars
issub(a::UnionT, b::Var, env) = issub_union(b, a, env, false, env.Lunions)
```

```
issub(a::Var, b::UnionT, env) = issub_union(a, b, env, true, env.Runions)

function issub(a::TagT, b::TagT, env)
  a === b && return true
  b === AnyT && return true
  a === AnyT && return false
  if a.name !== b.name
    a.name === TupleName && return false
    return issub(super(a), b, env)
  end
  if a.name === TupleName
    va, vb = a.vararg, b.vararg
    la, lb = length(a.params), length(b.params)
    if va && (!vb || la < lb)
      return false
    end
    ai = bi = 1
    while true
      ai > la && return bi > lb || (bi==lb && vb)
      bi > lb && return false
      !issub(a.params[ai], b.params[bi], env) && return false
      ai==la && bi==lb && va && vb && return true
      if ai < la || !va
        ai += 1
      end
      if bi < lb || !vb
        bi += 1
      end end
  else
    for i = 1:length(a.params)
      ai, bi = a.params[i], b.params[i]
      (issub(ai, bi, env) && issub(bi, ai, env)) || return false
    end end
  return true
end

function join(a,b,env)
  (a===BottomT || b===AnyT || a === b) && return b
  (b===BottomT || a===AnyT) && return a
  UnionT(a,b)
end

issub(a::Ty, b::Var, env) = var_gt(b, a, env)
issub(a::Var, b::Ty, env) = var_lt(a, b, env)
function issub(a::Var, b::Var, env)
```

```
  a === b && return true
  aa = env.vars[a]; bb = env.vars[b]
  if aa.right
    bb.right && return issub(bb.ub, bb.lb, env)
    return var_lt(a, b, env)
  else
    if !bb.right   # check   a ,b . a<:b
      return issub(aa.ub, b, env) || issub(a, bb.lb, env)
    end
    return var_gt(b, a, env)
  end
end

function var_lt(b::Var, a::Union(Ty,Var), env)
  bb = env.vars[b]
  !bb.right && return issub(bb.ub, a, env)  # check ∀b . b<:a
  !issub(bb.lb, a, env) && return false
  # for contravariance we would need to compute a meet here, but
  # because of invariance bb.ub ⊓ a == a here always.
  bb.ub = a  # meet(bb.ub, a)
  return true
end

function var_gt(b::Var, a::Union(Ty,Var), env)
  bb = env.vars[b]
  !bb.right && return issub(a, bb.lb, env)  # check ∀b . b>:a
  !issub(a, bb.ub, env) && return false
  bb.lb = join(bb.lb, a, env)
  return true
end

function issub_unionall(t::Ty, u::UnionAllT, env, R)
  haskey(env.vars, u.var) && (u = rename(u))
  env.vars[u.var] = Bounds(u.var.lb, u.var.ub, R)
  ans = R ? issub(t, u.T, env) : issub(u.T, t, env)
  delete!(env.vars, u.var)
  return ans
end

issub(a::UnionAllT, b::UnionAllT, env) = issub_unionall(a, b, env, true)
issub(a::UnionT, b::UnionAllT, env)    = issub_unionall(a, b, env, true)
issub(a::UnionAllT, b::UnionT, env)    = issub_unionall(b, a, env, false)
issub(a::Ty, b::UnionAllT, env)        = issub_unionall(a, b, env, true)
issub(a::UnionAllT, b::Ty, env)        = issub_unionall(b, a, env, false)
```

Bibliography

[1] Abelson, H., Dybvig, R. K., Haynes, C. T., Rozas, G. J., Adams, IV, N. I., Friedman, D. P., Kohlbecker, E., Steele, Jr., G. L., Bartley, D. H., Halstead, R., Oxley, D., Sussman, G. J., Brooks, G., Hanson, C., Pitman, K. M., and Wand, M. Revised report on the algorithmic language scheme. *SIGPLAN Lisp Pointers IV* (July 1991), 1–55.

[2] Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele, Jr., G. L., and Tobin-Hochstadt, S. The fortress language specification version 1.0. Tech. rep., March 2008.

[3] Baker, H. G. Equal rights for functional objects or, the more things change, the more they are the same. Tech. rep., Tech. Rept., Nimble Comp, 1992.

[4] Bolz, C. F., Diekmann, L., and Tratt, L. Storage strategies for collections in dynamically typed languages. In *Proc. 2013 ACM SIGPLAN Int. Conf. Object oriented Program. Syst. Lang. Appl. - OOPSLA '13* (New York, New York, USA, 2013), ACM Press, pp. 167–182.

[5] Bruce, K., Cardelli, L., Castagna, G., Leavens, G. T., and Pierce, B. On binary methods. *Theor. Pract. Object Syst. 1*, 3 (Dec. 1995), 221–242.

[6] Cardelli, L. *A Polymorphic λ-calculus with Type:Type.* Digital Systems Research Center, 1986.

[7] Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec. 1985), 471–523.

[8] Castagna, G., Ghelli, G., and Longo, G. A calculus for overloaded functions with subtyping. *Inf. Comput. 117*, 1 (Feb. 1995), 115–135.

[9] Castagna, G., Nguyen, K., Xu, Z., Im, H., Lenglet, S., and Padovani, L. Polymorphic functions with set-theoretic types. part 1: Syntax, semantics, and evaluation. In *POPL '14, 41th ACM Symposium on Principles of Programming Languages* (jan 2014), pp. 5–17.

[10] Castagna, G., and Pierce, B. C. Decidable bounded quantification. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1994), POPL '94, ACM, pp. 151–162.

[11] community, J. Parametric color types, 2014. `http://github.com/JuliaLang/Color.jl/issues/42`.

[12] Day, M., Gruber, R., Liskov, B., and Myers, A. C. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 1995), OOPSLA '95, ACM, pp. 156–168.

[13] Demmel, J. W., Dongarra, J. J., Parlett, B. N., Kahan, W., Gu, M., Bindel, D. S., Hida, Y., Li, X. S., Marques, O. A., Riedy, E. J., Vomel, C., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., Langou, J., and Tomov, S. Prospectus for the next LAPACK and ScaLAPACK libraries. Tech. Rep. 181, LAPACK Working Note, Feb. 2007.

[14] Dunfield, J. Elaborating intersection and union types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2012), ICFP '12, ACM, pp. 17–28.

[15] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., and Franz, M. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 465–478.

[16] Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J. G., and Willcock, J. A comparative study of language support for generic programming. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications* (New York, NY, USA, 2003), OOPSLA '03, ACM, pp. 115–134.

[17] Glew, N., Sweeney, T., and Petersen, L. A multivalued language with a dependent type system. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming* (New York, NY, USA, 2013), DTP '13, ACM, pp. 25–36.

[18] Gomez, C., Ed. *Engineering and Scientific Computing With Scilab*. Birkhäuser, 1999.

[19] Igarashi, A., and Nagira, H. Union types for object-oriented programming. In *Proceedings of the 2006 ACM Symposium on Applied Computing* (New York, NY, USA, 2006), SAC '06, ACM, pp. 1435–1441.

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

[21] Kell, S. In search of types. *Proc. 2014 ACM Int. Symp. New Ideas, New Paradig. Reflections Program. Softw. - Onward! '14* (2014), 227–241.

[22] Leavens, G. T., and Millstein, T. D. Multiple dispatch as dispatch on tuples. *ACM SIGPLAN Not. 33*, 10 (Oct. 1998), 374–387.

[23] mathematica. `http://www.mathematica.com`.

[24] MathWorks, Inc. *MATLAB Manual: mldivide*, r2014b ed. Natick, MA.

[25] matlab. `http://www.mathworks.com`.

[26] Morris, J. H. J. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[27] Murphy, M. Octave: A free, high-level language for mathematics. *Linux J. 1997* (July 1997).

[28] Pierce, B. Bounded quantification is undecidable. *Information and Computation 112*, 1 (1994), 131 – 165.

[29] Reynolds, J. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, N. Jones, Ed., vol. 94 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1980, pp. 211–258.

[30] Ronchi Della Rocca, S. Principal type scheme and unification for intersection type discipline. *Theor. Comput. Sci. 59*, 1-2 (July 1988), 181–209.

[31] Smith, D., and Cartwright, R. Java type inference is broken: Can we fix it? In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications* (New York, NY, USA, 2008), OOPSLA '08, ACM, pp. 505–524.

[32] van der Walt, S., Colbert, S. C., and Varoquaux, G. The numpy array: a structure for efficient numerical computation. *CoRR abs/1102.1523* (2011).