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

thank Tim Holy for frequently and spontaneously writing explanations that I found myself wanting to copy and paste into this writeup.

Contents

Chapter 1

Introduction

Much of the history of programming languages has been about increasing abstraction and generalization. Today we take it for granted that a single suitably powerful language could be used for nearly any programming task, were it not for pragmatic factors beyond our control. Over time, many special-purpose languages have been subsumed by these more powerful and general languages. The field of technical computing — programming for applied math and the sciences — is something of a holdout from this trend, with its own purpose-built languages still dominating. There are probably many reasons for this, including historical, social and technical.

In this work we will focus on the technical reasons for this, and the problems that the current state of affairs causes.

What do we mean by a *language* for scientific computing? The prevailing answer is that it is about numeric arrays, perhaps matrices specifically, and numerical libraries that use them. Instead we intend to argue that, at a deeper level, it is about certain kinds of flexibility, particularly flexibility in the behavior of key operators and functions. This flexibility is needed to maximize composability and reusability in scientific code. Without it, certain programs may be easy to write today, but changing functional and performance requirements can become difficult to meet.

A new programming language is not the only thing the world of technical computing needs. Compiler techniques, library design, high-performance computational kernels, and approaches to parallelism are all important, and can usually be applied within multiple languages. For example the C and Fortran language families have taken on many changes and extensions over the years to accomodate new technologies. So we do not expect a new language to "solve" technical computing all by itself, but rather we want to ask whether any language-level abstractions can provide a better base for high-productivity technical computing.

It is clear that scientific and technical users have their own particular needs. So far there have been two paths available for fulfilling these needs: using a mainstream general-purpose language like C++ or Java, or designing a new language like R or MATLAB. Designing your own language seems like a drastic thing to do, and yet it is unusually common in technical computing (see gang of 40).

1.1 The technical computing problem

Figure **??** 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 [7] 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 [1] or Polyduce [5], 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).

Our contribution derives from experience with the second approach. 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, various partial evaluations can be done: generating code without type checks, removing

[1] For an excellent discussion of the many meanings of the word "type" see [12].

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 [2]. If this information is not reflected in the source language, then certain uses like passing data to native code become unnecessarily more complicated.

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.

past (static) type systems for dispatch were designed to ensure the absence of no-method errors and ambiguities (completeness and uniqueness). our goal is instead to statically resolve methods. this is inherently heuristic and best-effort. since static types shouldn't affect program behavior, we conclude that the dispatch must be dynamic, which is happily the same conclusion you would reach if you simply wanted dynamic typing.

interesting variants: - require static single method matches - reject programs with no-method errors - reject programs that yield Unions

By *selection* we mean any mechanism used to pick one of several pieces of code to run. This includes object-oriented mechanisms, as well as function overloading, and even branches.

By *specialization* we mean transformation of a given piece of code, particularly as done by a compiler to generate efficient code for some special case of a flexible piece of high-level code.

Specialization entails program analysis. By necessity, a language's code selection mechanisms must inform this process, to ensure the correctness of the analysis. A central argument of this thesis is that specialization should feed back into selection, allowing the *approximate values* processed by a compiler to be used in the source language for dispatch. We argue that this is the key feature missing from the present generation of dynamically-typed languages.

Solution: integrate code specialization and code selection

Why integrate code specialization and code selection?

- specialization requires selection anyway
- simpler system
- can sometimes collapse 2 layers of dispatch into 1
- can replace library code with a code generator without either changing client code *or* any extra overhead

Extensive code specialization is a key feature of technical computing. "what to specialize on" has been an open problem. our types are a possible solution to this for two reasons:

1. you can tune the amount of information they contain
2. everybody agrees to use them, which helps ensure that specializing on the types will actually do something useful.

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." [16]

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

At a high level, the experiences and thoughts that give rise to our design decisions.

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.

2.2 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

2.3 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

2.3.1 A compiler for every problem

2.4 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 1x1 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*.

2.4.1 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.

2.4.2 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`.

2.4.3 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.

2.4.4 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. numerical properties (such as positive, negative, even, odd, greater than 1, etc.) are what matter, and these are highly dynamic. the lattices involved are generally not of finite height.

different number representations exist only for efficiency, and are often incidental to the meaning of a piece of code. for example people want "y = sqrt(x)" to compute the square root of x whether it is positive or negative, and give a real or complex result accordingly. as another example, in many cases it is convenient not to need to worry about integer overflow.

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.

2.5 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 bro-

ken 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.

How does Julia break through the 4x-slower-than-C performance barrier?

bits types and parameters – can have abstract types whose size is known uncondi-

tionally

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

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 would 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 [3] 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
    for a in animals
        b = nearby_animal(a)
```

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

```
        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 further. 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, antisymmetric, 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 **??** 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 definitions.

3.2.2 Symbolic programming

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 (TODO cite). However, they seem to be decidable for most practical programs, and also admit minor

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 "really" 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.

variations that yield decidable systems (TODO cite). It is fair to say they are *just barely* undecidable, while predicates are *very* undecidable.

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 nonterminating 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.

3.4 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.

Joins of quantified types

"Diagonal dispatch"

(T,T) (Int, Real)

\(Int,Int)/

Avoid hard coding logic into the compiler.

Exposing to the user what is part of the language anyway and not just hiding it in the compiler.

Data flow analysis has to actually work

Taking things out of the language that break dataflow analysis

Local reasoning

You are imposing a type system anyway, even implicitly perhaps, so might as well make that from the get-go.

3.4.1 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 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 metasymbols 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 subexpressions, 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* for reasons that have been addressed in the literature [6]. Invariance preserves the property that the only subtypes of a concrete type are `Bottom` and itself. We also find that most uses of covariance are more flexibly handled by union type connectives, which will be 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:

$$[\![\texttt{Tuple}\{P_1, \cdots \quad, P_n\}]\!] = \prod_{1 \le i \le n} [\![P_i]\!]$$

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

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 [14]. 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, 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. `Union` types also commute with covariant type constructors (tuples in our case), providing even more ways to rewrite types:

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

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 [9] [18] (TODO cite analyses that have used union types).

The next problem we need to solve arises from combining data-flow analysis and

parametric invariance. 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 if `C` is invariant. 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 [4]; 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.

4.1.1 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.

4.1.2 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.

4.1.3 Associated types and type computation

4.1.4 Subtyping

Describe algorithm.

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 logical 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 `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).

4.1.5 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
- lower bounds in quantifiers
- arbitrary predicates, theory of natural numbers, etc.

4.2 Dispatch mechanism

4.3 Data model

4.4 Performance model

4.4.1 Type inference

4.4.2 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}(::Type{T}, x::T) = 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 any type less than it in the second argument slot.

- The abstractions of equality and comparison. Different equivalence classes between is/===, isequal and ==
- Numeric vs lexicographic ordering?

cmp, lexcmp, vs isless, ¡

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").

Two further problems should also be kept in mind. First, the natural isomorphisms between sets of numbers might not be isomorphisms on a real computer. For example, due to the behavior of floating-point arithmetic, an operation on complex numbers with zero imaginary part might not give an answer equal to the same operation on real numbers. Second, the contexts that demand use of one type of number or another are often not easily described by type systems. 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 — should a type system attempt to prove a matrix symmetric before we compute its eigenvalues? While we cannot offer a once-and-for-all solution to these problems, we will show how the flexibility of our proposed mechanism is useful for addressing them.

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.

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 become easy to obtain, people find more and more uses for them.

Ordinal types: Pointer, Char

Integer types: Int8, Int16, Int32, Int64, Int128, UInt8, UInt16, UInt32, UInt64, UInt128, BigInt

Floating point types: Float16, Float32, Float64, BigFloat (Float128, double-double)

Fixed point: Fixed32b, Ufixed8, Ufixed16

Extensions: Complex, Quaternion, Interval, DualNumber

Number-like: Date, TimePeriod, DatePeriod, Color, (Units), DNA nucleotide type (bioseq.jl) dates: different cardinal and ordinal behavior

musical notes

Applications

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.

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.

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. E.g. in images, each dimension might be quite different, e.g. time vs. space vs. color, so you don't want to drop or rearrange dimensions very often.

Here are our ground rules:

1. You can't manually implement the behavior inside the compiler

2. The compiler must be able to reasonably understand the program

3. The code must be reasonably easy to write

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 net-better solution.

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.

Here is a sample derivation for the call `index_shape(1:m,1,1:n)` (the argument type tuple is `(Range1,Int,Range1)`), using the first definition above (dropping scalar-indexed dimensions):

```
index_shape(1:n, 1, 1:m) => tuple(length(::Range1)::Int, index_shape((::Int, ::Range1)..

index_shape((::Int, ::Range1)...) => index_shape(::Int, ::Range1)

index_shape(::Int, ::Range1) => index_shape((::Range1,)...)

index_shape((::Range1,)...) => index_shape(::Range1)

index_shape(::Range1) => tuple(length(::Range1)::Int, index_shape(()...)...)

index_shape(()...) => index_shape()::()

back substitute => tuple(length(::Range1)::Int, index_shape(()...)::()...)::(Int,)
```

```
back substitute => tuple(length(::Range1)::Int, ::(Int,)...)

tuple(::Int, ::(Int,)...) => tuple(::Int, ::Int)

::(Int, Int)
```

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).

```
diverge() = randbool() ? () : tuple(1, diverge()...)
```

5.1.4 Array views

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

Multiple dispatch on special matrices

29 LAPACK types via composition of 9 types, issue 8240

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.

Features of julia this demonstrates:

- functions that need to dispatch on more than one argument - staged methods providing a natural way to integrate code generation - doing specialization through the object system, leading to a "flat" system

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: $L_{mn} =: \langle b_m, \mathcal{L}b_n \rangle = \int b_m(x)K(x-x')b_n(x')dx \quad dx'$. 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)

5.8 misc staged functions

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."

Chapter 6

Wholesale copy of SIAM REVIEW PAPER

6.1 Scientific computing languages: The Julia innovation

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 [**?**], R [10], Mathematica [**?**], Octave [17], Python (with NumPy) [19], and SciLab [8] 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.

Julia is dynamically typed, but it is different as it approaches statically typed performance. New users can begin working with Julia as they did in the traditional numerical computing languages, and work their way up when ready. In Julia, types are implied by the computation itself together with input values. As a result, Julia programs are often completely generic and compute with data of different input types without modification—a feature known as "polymorphism."

It is a fact that compilers need type information to emit efficient code for high performance. It is also true that the convenience of not typing gives dynamic systems a major advantage in productivity. Accordingly, many researchers today do their day-to-day work in dynamic languages. Still, C and Fortran remain the gold standard for computationally-intensive problems because high-level dynamic languages have lacked performance.

An unfortunate outcome of the currently popular languages is that the most challenging areas of numerical computing have benefited the least from the increased abstraction and productivity offered by higher level languages. A pervasive idea that Julia wishes to shatter is that numerical computing languages are "prototyping" languages and one must switch to C or Fortran for performance. The consequences to scientific computing have been more serious than many realize.

We recommend signing up at `juliabox.org` to use Julia conveniently through the Julia prompt or the IJulia notebook. (We like to say "download,next,next,next" is three clicks too many!) If you wish to, you can download Julia at `www.julialang.org`, along with the IJulia notebook.

We wish to show how users can be coaxed into writing better software that is comfortably familiar, easier to work with, easier to maintain, and can perform very well. This paper brings together

- Examples to orient the new user and

- The theory and principles that underly Julia's innovative design

specifically so that everyday users of numerical computing languages may understand why Julia represents a truly fresh approach.

6.1.1 Computing Research Transcends Communities

Numerical computing research has always lived on the boundary of computer science, engineering, mathematics, and computational sciences. Readers might enjoy trying to label the "Top 10 algorithms"[**?**] by field, and may quickly see that advances typically transcend any one field with broader impacts to science and technology as a whole.

Computing is more than using an overgrown calculator. It is a cross cutting communication medium. Research into programming languages therefore breaks us out of our research boundaries.

The first decade of the 21st century saw a boost in such research with the High Productivity Computing Systems DARPA funded projects into the languages Chapel [**?**], Fortress [**?**] and X10 [**?**].

Also contemporaneous has been a growing acceptance of Python. Up to around 2009 some of us were working on the Star-P interactive high performance computing system for parallelizing primarily MATLAB but also Python, R, Excel and other high level languages were researched. (User Guide: [**?**], Getting Started: [**?**], Papers [**?**, **?**, **?**, **?**]). Julia continues our research into parallel computing.

More recently Google, Apple, and Mozilla have introduced Go [**?**], Swift [**?**], and Rust [**?**] respectively as new interesting dynamic languages which are gaining traction.

6.2 A taste of Julia

6.2.1 A brief tour

Since Julia is a young language, new users may worry that it is somehow immature or lacking some functionality. Our experience shows that Julia is rapidly heading towards maturity. Most new users are surprised to learn that Julia has much of the functionality of the traditional established languages, and a great deal of functionality not found in the older languages.

New users also want a quick explanation as to why Julia is fast, and whether somehow the same "magic dust" could also be sprinkled on their traditional scientific computing language. Why Julia is fast is a combination of many technologies some of which are introduced in Sections 6.4 through 6.6 in this paper. Julia is fast because we, the designers, developed it that way for us, the users. Performance is fragile, like accuracy, one arithmetic error can ruin an entire otherwise correct computation. We do not believe that a language can be designed for the human, and easily retrofitted for the computer. Rather a language must be designed from the start for the human and the computer. Section 6.3.1 and [**?**] explores this issue.

Before we get into the details of Julia, we provide a brief tour for the reader to get a beginners' feel for the language. Users of other dynamic numerical computing environments may find some of the syntax familiar. In language design, there is always a trade-off between making users feel comfortable in a new language vs. clean language design. In Julia, we maintain superficial similarity to existing environments to ease new users, but not at the cost of sacrificing good language design or performance.

```
In[1]: A = rand(3,3) + eye(3) # Familiar Syntax
       inv(A)
```

```
Out[1]: 3x3 Array{Float64,2}:
         0.698106 -0.393074 -0.0480912
        -0.223584  0.819635 -0.124946
        -0.344861  0.134927  0.601952
```

The output from the Julia prompt says that A is a two dimensional matrix of size 3×3, and contains double precision floating point numbers.

Indexing of arrays is performed with brackets, and is 1-based. It is also possible to compute an entire array expression and then index into it, without assigning the expression to a variable:

```
In[2]: x = A[1,2]
       y = (A+2I)[3,3] # The [3,3] entry of A+2I
```

```
Out[2]: 2.601952
```

In Julia, `I` is a built-in representation of the identity matrix, which can be scaled and combined with other matrix operations, without explicitly forming the identity matrix as is commonly done using commands such as "eye" which unnecessarily requires $O(n^2)$ storage and unnecessary computation in traditional languages and in the end is a cute but ultimately wrong word for the identity matrix. The built-in representation is fully implemented in Julia code in the Julia base library.

Julia has symmetric tridiagonal matrices as a special type. For example, we may define Gil Strang's favorite matrix (the second order difference matrix) in a way that uses only $O(n)$ memory.

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

```
Out[3]: 7x7 SymTridiagonal{Float64}:
```

```
 2.0  -1.0   0.0   0.0   0.0   0.0   0.0
-1.0   2.0  -1.0   0.0   0.0   0.0   0.0
 0.0  -1.0   2.0  -1.0   0.0   0.0   0.0
 0.0   0.0  -1.0   2.0  -1.0   0.0   0.0
 0.0   0.0   0.0  -1.0   2.0  -1.0   0.0
 0.0   0.0   0.0   0.0  -1.0   2.0  -1.0
 0.0   0.0   0.0   0.0   0.0  -1.0   2.0
```

Julia calls an efficient $O(n)$ tridiagonal solver:

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

```
Out[4]: 8-element Array{Float64,1}:
```

```
 4.0
 7.0
 9.0
10.0
10.0
 9.0
 7.0
 4.0
```

Consider the sorting of complex numbers. Sometimes it is handy to have a sort that generalizes the real sort. This can be done by sorting first by the real part, and where there are ties, sort by the imaginary part. Other times it is handy to use the polar representation, which sorts by radius than angle.

If a numerical computing language "hard-wires" its sort to be one or the other, it

misses a wonderful opportunity. A sorting algorithm need not depend on details of what is being compared or how it is being compared. One can abstract away these details thereby reusing a sorting algorithm for many different situations. One can specialize later. Thus alphabetizing strings, sorting real numbers, or sorting complex numbers in two or more ways all run with the same code.

In Julia, one can turn a complex number `w` into an ordered pair of real numbers (a tuple of length 2) such as the Cartesian form `(real(w),imag(w))` or the polar form `(abs(w),angle(w))`. Tuples are then compared lexicographically in Julia. The sort command takes an optional "less-than" operator, `lt`, which is used to compare elements when sorting.

In[5]:
```
# Cartesian comparison sort of complex numbers
complex_compare1(w,z) = (real(w),imag(w)) <
(real(z),imag(z))
sort([-2,2,-1,im,1], lt = complex_compare1 )
```

Out[5]:
```
5-element Array{Complex{Int64},1}:
-2+0im
-1+0im
 0+1im
 1+0im
 2+0im
```

In[6]:
```
# Polar comparison sort of complex numbers
complex_compare2(w,z) = (abs(w),angle(w)) <
(abs(z),angle(z))
sort([-2,2,-1,im,1], lt = complex_compare2)
```

```
Out[6]: 5-element Array{Complex{Int64},1}:
         1+0im
         0+1im
        -1+0im
         2+0im
        -2+0im
```

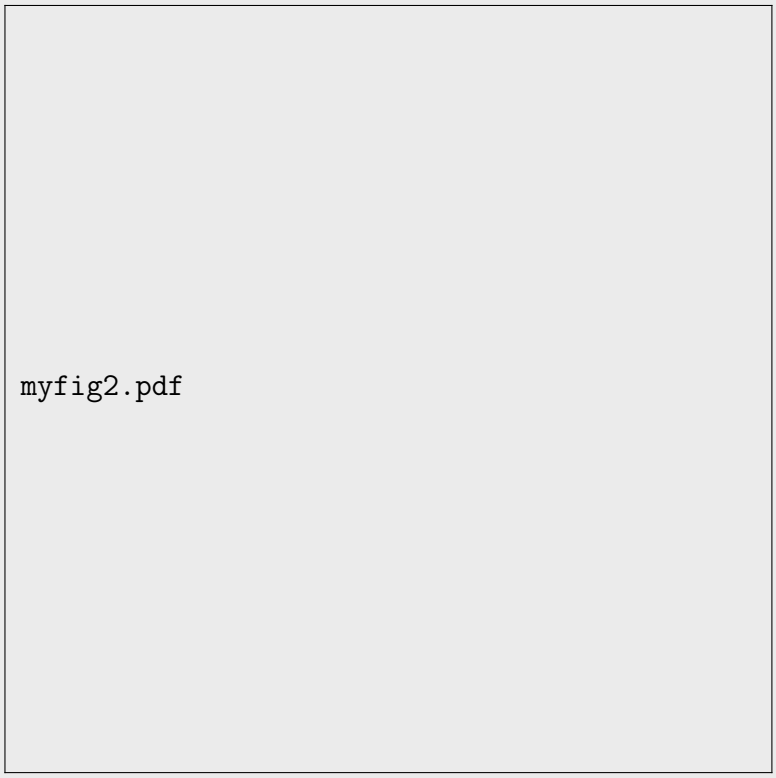
Note the `Array{ElementType,dims}` syntax. It shows up everywhere. In the above example, the elements are complex numbers whose parts are `Int64`'s. The `1` indicates it is a one dimensional vector.

To be sure, experienced computer scientists tend to suspect there is nothing new under the sun. The C function `qsort()` takes a `compar` function. Nothing really new there. Python also has custom sorting with a key. MATLAB's sort is more basic. The real contribution of Julia, as will be fleshed out further in this paper, is that the design of Julia allows custom sorting to be much faster than other dynamic languages.

The next example that we have chosen for the introductory taste of Julia is a quick plot of Brownian motion. This example uses the matplotlib python package for graphics.

In[7]:
```
Pkg.add("PyPlot") # Download the PyPlot package
using PyPlot # load the functionality into Julia

for i=1:5
    y=cumsum(randn(500))
    plot(y)
end
```



The matplotlib package is popular for users coming from Python or MATLAB. Gadfly is another very popular package for plotting. Gadfly was built by Daniel Jones and was influenced by the well admired Grammar of Graphics (see [**?**] and [**?**]).

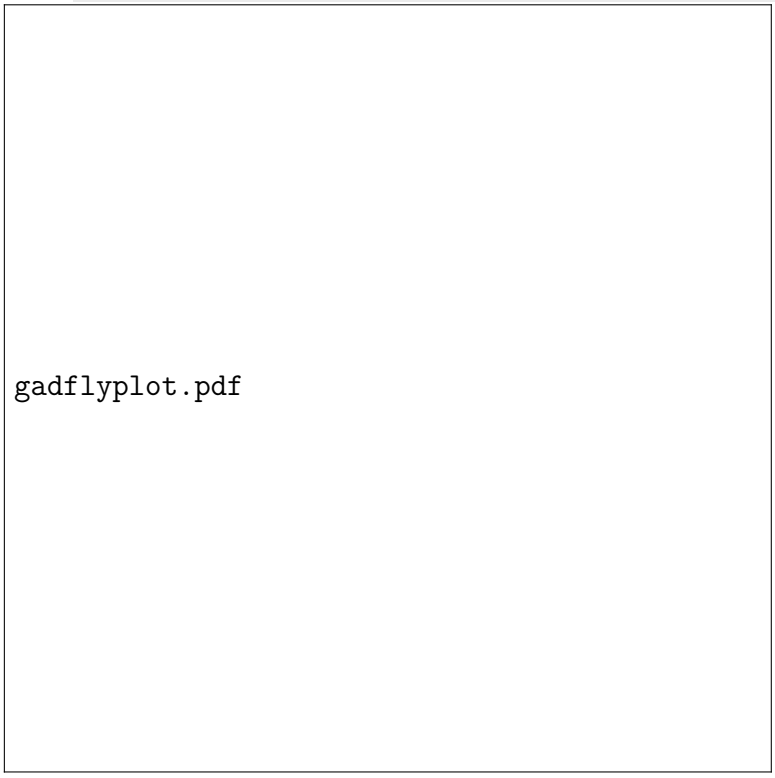
The syntax might be less familiar to some users of numerical computing languages, but it is not hard to learn.[1] Many Julia users find Gadfly more flexible and prefer the look of the output. Gadfly has also added to the building of beautiful interactive graphics through the `Interact` package. See `https://github.com/JuliaLang/Interact.jl/issues/36` for examples of Interact.

[1] See tutorial on `http://gadflyjl.org`

In[8]:

```
Pkg.add("Gadfly") # Download the Gadfly package
using Gadfly # load the functionality into Julia


n = 500
p = [layer(x=1:n, y=cumsum(randn(n)), color=[i], Geom.line)
    for i in ["First","Second","Third"]]
labels=(Guide.xlabel("Time"),Guide.ylabel("Value"),
        Guide.Title("Brownian Motion Trials"),Guide.colorkey("Trial"))
plot(p...,labels...)
```



In our last example in this introductory tour, we illustrates graphics and histogramming: a user performs a simple Monte-Carlo experiment: the famous semicircle law from Random Matrix Theory for a random eigenvalue of a symmetric matrix. In Section 6.6.3

we provide examples of parallel monte carlo histograms.

In[9]:

```
n=100 # Matrix Size
t=1000 # Number of Trials
sym(A)=A+A' # Define a function named sym to symmetrize a
matrix

# Eigenvalues of t nxn matrices saved as vector
z= [[eigvals(sym(randn(n,n))) for i=1:t]...]
z/=sqrt(2n)

# Histogram Plot
# using PyPlot
plt.figure(figsize=(8,3.5))
x=-2:.01:2;

plot(x,sqrt(4.-x.^2)/(2π),"r")
plt.hist(z,bins=50,normed=true)


# Label and save to file
axis([-3,3,0,.4])

xlabel("normalized λ")
ylabel("pdf")
title("Wigner Semi-Circle Law")
plt.savefig("myfig.pdf")
```

Julia has been in development since 2009; a public release was announced in February of 2012. It is an active open source project with over 250 contributors and is available under the MIT License [**?**] for open source software. Nurtured at the Massachusetts Institute of Technology, but with contributors from around the world, Julia is increasingly seen as a high-performance alternative to R, Matlab, Octave, Python, and SciLab, or a high-productivity alternative to C, C++ and Fortran. It is also recognized as being better suited to general purpose computing tasks than traditional numerical computing systems, allowing it to be used not only to prototype numerical algorithms, but also to deploy those algorithms, and even serve results of numerical computations to the rest of the world.[2] Perhaps most significantly, a rapidly growing ecosystem of high-quality, open source, composable numerical packages (over 450 in number, see `http://pkg.julialang.org` and especially `http://pkg.julialang.org/pulse.html`) written in Julia has emerged, including libraries for linear algebra, statistics, optimization, data analysis, machine learning and many other applications of numerical computing.

6.2.2 An invaluable tool for numerical integrity

One popular feature of Julia is that it gives the user the ability to "kick the tires" of a numerical computation. We thank Velvel Kahan for the sage advice[3] concerning the importance of this feature.

The idea is simple: a good engineer tests his or her code for numerical stability. In Julia this can be done by changing IEEE rounding modes. There are five modes to choose from, yet most engineers silently only choose the `RoundNearest` mode default available in many numerical computing systems. If a difference is detected, one can also run the computation in higher precision. Kahan writes

[2]Sudoku as a service, by Iain Dunning, `http://iaindunning.com/2013/sudoku-as-a-service.html`, is a wonderful example where a Sudoku puzzle is solved using the optimization capabilities of the JUMP.jl Julia package [**?**] and made available as a web service.

[3]Personal communication, January 2013, in the Kahan home, Berkeley, California

> Can the effects of roundoff upon a floating-point computation be assessed without submitting it to a mathematically rigorous and (if feasible at all) time-consuming error-analysis? In general, No.
>
> ...
>
> Though far from foolproof, rounding every inexact arithmetic operation (but not constants) in the same direction for each of two or three directions besides the default To Nearest is very likely to confirm accidentally exposed hypersensitivity to roundoff. When feasible, this scheme offers the best *Benefit/Cost* ratio. [**?**]

As an example, we round a 15x15 Hilbert-like matrix, and take the [1,1] entry of the inverse computed in various round off modes. The radically different answers dramatically indicates the numerical sensitivity to roundoff. We even noticed that slight changes to LAPACK give radically different answers. Very likely you will see different numbers when you run this code due to the very high sensitivity to roundoff errors.

In[10]:
```
h(n)=[1/(i+j+1) for i=1:n,j=1:n]
```

Out[10]:
```
h (generic function with 1 method)
```

In[11]:
```
H=h(15);
with_rounding(Float64,RoundNearest) do
    inv(H)[1,1]
end
```

Out[11]:
```
154410.55589294434
```

In[12]:
```
with_rounding(Float64,RoundUp) do
    inv(H)[1,1]
end
```

Out[12]:
```
-49499.606132507324
```

```
In[13]: with_rounding(Float64,RoundDown) do
            inv(H)[1,1]
        end
```

```
Out[13]: -841819.4371948242
```

With 300 bits of precision we obtain

```
In[14]: with_bigfloat_precision(300) do
            inv(big(H))[1,1]
        end
```

```
Out[14]: -2.093971792507462701282801742144895161627088577037149597632326890471535076588249105499837625 2e+03
```

Note this is the [1,1] entry of the inverse of the rounded Hilbert-like matrix, not the inverse of the exact Hilbert-like matrix. Also, the results are senstive to the BLAS and LAPACK, and the results may differ on different machines with different versions of Julia.

6.2.3 Julia architecture and language design philosophy

Many popular dynamic languages were not designed with the goal of high performance. After all, if you wanted really good performance you would use a static language, or so the popular wisdom would say. Only with the increased need in the day-to-day life of scientific programmers for simultaneous productivity and performance in a single system has the need for high-performance dynamic languages become pressing. Unfortunately, retrofitting an existing slow dynamic language for high performance is almost impossible *specifically* in numerical computing ecosystems. This is because numerical computing requires performance-critical numerical libraries, which invariably depend on the details of the internal implementation of the high-level language, thereby locking in those internal

implementation details. For example, you can run Python code much faster than the standard CPython implementation using the PyPy just-in-time compiler; but PyPy is currently incompatible with NumPy and the rest of SciPy.

Another important point is that just because a program is available in C or Fortran, it may not run efficiently from the high level language or be easy to "glue" it in. For example when Steven Johnson tried to include his C erf function in Python he reported that Pauli Virtane had to write glue code[4] to vectorize the erf function over the native structures in Python in order to realize the performance and Johnson had to write glue code for Matlab, Octave, and Scilab. The Julia effort was, by contrast, effortless.[5] As another example, a benchmark of the normal random number generator written in pure Julia is twice as fast as MATLAB's `randn` written in a lower level language.

The best path to a fast, high-level system for scientific and numerical computing is to make the system fast enough that all of its libraries can be written in the high-level language in the first place. The JUMP.jl library [**?**] for mathematical optimization written by Miles Lubin and Iain Dunning is a great example of the success of this approach—the entire library is written in Julia and uses many language features such as metaprogramming, user-defined parametric types, and multiple dispatch extensively to provide a seamless interface to describe various kinds of optimization problems and solve them with any number of commercially or freely available solvers.

The Two Language Problem: As long as the developers' language is harder than the users' language, numerical computing will always be hindered

This is an essential part of the design philosophy of Julia: all basic functionality must be possible to implement in Julia—never force the programmer to resort to using C or Fortran. Julia solves the two language problem. Basic functionality must be fast: integer arithmetic, for loops, recursion, floating-point operations, calling C functions,

[4] `https://github.com/scipy/scipy/commit/ed14bf0`

[5] Steven Johnson, personal communication. See `http://ab-initio.mit.edu/wiki/index.php/Faddeeva_Package`)

manipulating C-like structs. While these are not only important for numerical programs, without them, you certainly cannot write fast numerical code. "Vectorization languages" like Matlab, R, and Python+NumPy hide their for loops and integer operations, but they are still there, inside the C and Fortran, lurking behind the thin veneer. Julia removes this separation entirely, allowing high-level code to "just write a for loop" if that happens to be the best way to solve a problem.

We believe that the Julia programming language fulfills much of the Fortran dream: automatic translation of formulas into efficient executable code. It allows programmers to write clear, high-level, generic and abstract code that closely resembles mathematical formulas, as they have grown accustomed to in dynamic systems, yet produces fast, low-level machine code that has traditionally only been generated by static languages.

Julia's ability to combine these levels of performance and productivity in a single language stems from the choice of a number of features that work well with each other:

1. An expressive parametric type system, allowing optional type annotations; (See Section 6.3.3 for parametric types and Section 6.4.1 for optional type annotations.)

2. Multiple dispatch using those types to select implementations (Section 6.4);

3. A dynamic dataflow type inference algorithm allowing types of most expressions to be inferred (Section 6.3.5);

4. Careful design of the language and standard library to be amenable to type analysis (Section 6.6);

5. Aggressive code specialization against run-time types (Section 6.5);

6. Metaprogramming for sophisticated code generation (Section 6.5);

7. Just-In-Time (JIT) compilation using the Low-level Virtual Machine (LLVM) compiler framework [13], which is also used by a number of other compilers such as Clang [**?**] and Apple's Swift [**?**] (Section 6.5).

Although a sophisticated type system is made available to the programmer, it remains unobtrusive in the sense that one is never required to specify types, nor are type annotations necessary for performance. Type information flows naturally from having actual values (and hence types) during code generation, and from the multiple dispatch paradigm: by expressing polymorphic behavior of functions declaratively with multiple dispatch, the programmer provides the compiler with extensive type information, while also enjoying increased expressiveness.

In what follows, we describe the benefits of Julia's language design for numerical computing, allowing programmers to more readily express themselves and obtain performance at the same time.

6.3 Writing programs with and without types

6.3.1 The balance between human and the computer

Graydon Hoare, author of the Rust programming language [**?**], in an essay on "Interactive Scientific Computing" [**?**] defined programming languages succinctly:

> Programming languages are mediating devices, interfaces that try to strike a balance between human needs and computer needs. Implicit in that is the assumption that human and computer needs are equally important, or need mediating.

Catering to the needs of both the human and the computer is a repeating theme in this paper. A program consists of data and operations on data. Data is not just the input file, but everything that is held —an array, a list, a graph, a constant—during the life of the program. The more the computer knows about this data, the better it is at executing operations on that data. Types are exactly this metadata. Describing this metadata, the types, takes real effort for the human. Statically typed languages such as C and

Fortran are at one extreme, where all types must be defined and are statically checked during the compilation phase. The result is excellent performance. Dynamically typed languages dispense with type definitions, which leads to greater productivity, but lower performance as the compiler and the runtime cannot benefit from the type information that is essential to produce fast code. Can we strike a balance between the human's preference to avoid types and the computer's need to know?

6.3.2 Julia's recognizable types

Julia's design allows for the gradual learning of concepts, where users start in a manner that is familiar to them and over time, learn to structure programs in the "Julian way" (a term that captures well-structured readable high performance Julia code). Julia users coming from other numerical computing environments have a notion that data may be represented as matrices that may be dense, sparse, symmetric, triangular, or of some other kind. They may also, though not always, know that elements in these data structures may be single precision floating point numbers, double precision, or integers of a specific width. In more general cases, the elements within data structures may be other data structures. We introduce Julia's type system using matrices and their number types:

```
In[15]: rand(1,2,1)
```

```
Out[15]: 1x2x1 Array{Float64,3}:
[ :, :, 1] =
  0.789166 0.652002
```

```
In[16]: [1 2; 3 4]
```

```
Out[16]: 2x2 Array{Int64,2}:
  1 2
  3 4
```

```
In[17]: [true; false]
```

```
Out[17]: 2-element Array{Bool,1}:
           true
           false
```

We see a pattern in the examples above. `Array{T,ndims}` is the general form of the type of a dense array with `ndims` dimensions, whose elements themselves have a specific type `T`, which is of type double precision floating point in the first example, a 64-bit signed integer in the second, and a boolean in the third example. Therefore `Array{T,1}` is a 1-d vector (first class objects in Julia) with element type `T` and `Array{T,2}` is the type for 2-d matrices.

It is useful to think of arrays as a generic N-d object that may contain elements of any type `T`. Thus `T` is a type parameter for an array that can take on many different values. Similarly, the dimensionality of the array `ndims` is also a parameter for the array type. This generality makes it possible to create arrays of arrays. For example, Using Julia's array comprehension syntax, we create a 2-element vector containing 2×2 identity matrices.

```
In[18]: a = [eye(2) for i=1:2]
```

```
Out[18]: 2-element Array{Array{Float64,2},1}:
```

Many first time users are already familiar with basic floating point number types such as `Float32` (single precision) and `Float64` (double precision). In addition, Julia also provides a built-in `BigFloat` type that may interest a new Julia user. With `BigFloat`s, users can run computations in arbitrary precision arithmetic, thereby exploring the numerical properties of their computation.

6.3.3 User's own types are first class too

Many dynamic languages for numerical computing have traditionally had an asymmetry, where built-in types have much higher performance than any user-defined types. This is not the case with Julia, where there is no meaningful distinction between user-defined and "built-in" types.

We have mentioned so far a few number types and two matrix types, `Array{T,2}` the dense array, with element type `T` and `SymTridiagonal{T}`, the symmetric tridiagonal with element type `T`. There are also other matrix types, for other structures including SparseMatrixCSC (Compressed Sparse Columns), Hermitian, Triangular, Bidiagonal, and Diagonal. Julia's sparse matrix type has an added flexibility that it can go beyond storing just numbers as nonzeros, and instead store any other Julia type as well. The indices in SparseMatrixCSC can also be represented as integers of any width (16-bit, 32-bit or 64-bit). All these different matrix types, although available as built-in types to a user downloading Julia, are implemented completely in Julia, and are in no way any more or less special than any other types one may define in their own program.

For demonstration, we create a symmetric arrow matrix type that contains a diagonal

and the first row `A[1,2:n]`.

In[19]:
```
# Parametric Type Example (Parameter T)
# Define a Symmetric Arrow Matrix Type with elements of
type T

type SymArrow{T}
    dv::Vector{T} # diagonal
    ev::Vector{T} # 1st row[2:n]
end

# Create your first Symmetric Arrow Matrix
S = SymArrow([1,2,3,4,5],[6,7,8,9])
```

Out[19]:
```
SymArrow{Int64}([1,2,3,4,5],[6,7,8,9])
```

Our purpose here is to introduces "Parametric Types". Later in Section 6.4.4, we develop this example much further. The `SymArrow` matrix type contains two vectors, one each for the diagonal and the first row, and these vector contain elements of type `T`. In the type definition, the type `SymArrow` is parametrized by the type of the storage element `T`. By doing so, we have created a generic type, which refers to a universe of all arrow matrices containing elements of all types. The matrix `S`, is an example where `T` is `Int64`. When we write functions in Section 6.4.4 that operate on arrow matrices, those functions themselves will be generic and applicable to the entire universe of arrow matrices we have defined here.

Julia's type system allows for abstract types, concrete "bits" types, composite types, and immutable composite types. All of these can be parametric and users may even write programs using unions of these different types. We refer the reader to read all

about Julia's type system in the types chapter in the Julia manual[6].

6.3.4 Vectorization: Key Strengths and Serious Weaknesses

Users of traditional high level computing languages know that vectorization improves performance. Do most users know exactly why vectorization is so useful? It is precisely because, by vectorizing, the user has promised the computer that the type of an entire vector of data matches the very first element. This is an example where users are willing to provide type information to the computer without even knowing exactly that is what they are doing. Hence, it is an example of a strategy that balances the computer's needs with the human's.

From the computer's viewpoint, vectorization means that operations on data happen largely in sections of the code where types are known to the runtime system. When the runtime is operating on arrays, it has no idea about the data contained in an array until it encounters the array. Once encountered, the type of the data within the array is known, and this knowledge is used to execute an appropriate high performance kernel. Of course what really occurs at runtime is that the system figures out the type, and gets to reuse that information through the length of the array. As long as the array is not too small, all the extra work in gathering type information and acting upon it at runtime is amortized over the entire operation.

The downside of this approach is that the user can achieve high performance only with built-in types, and user defined types end up being dramatically slower. The restructuring for vectorization is often unnatural, and at times not possible. We illustrate this with an

[6]See the chapter on types in the Julia manual: `http://docs.julialang.org/en/latest/manual/types/`

example of the cumulative sum computation.

In[20]:
```
# Sum prefix (cumsum) on vector w with elements of type T
function prefix{T}(w::Vector{T})
    for i=2:size(w,1)
       w[i]+=w[i-1]
    end
    w
end
```

We execute this code on a vector of double precision numbers and double-precision complex numbers and observe something that may seem remarkable: similar running times.

In[21]:
```
x = ones(1_000_000)
@time prefix(x)

y = ones(1_000_000) + im*ones(1_000_000)
@time prefix(y);
```

Out[21]:
```
elapsed time:  0.003243692 seconds (80 bytes allocated)
elapsed time:  0.003290693 seconds (80 bytes allocated)
```

This simple example is difficult to vectorize, and hence is often built into many numerical computing systems. In Julia, the implementation is very similar to the snippet of code above, and runs at speeds similar to C. While Julia users can write vectorized programs like in any other dynamic language, vectorization is not a pre-requisite for performance. This is because Julia strikes a different balance between the human and the computer when it comes to specifying types. Julia allows optional type annotations.

Generally, in Julia, type annotations are not for performance. They are purely for code selection. (See Section 6.4.) If the programmer annotates their program with types,

the Julia compiler will use that information. But for the most part, user code often includes minimal or no type annotations, and the Julia compiler automatically infers the types.

6.3.5 Type inference rescues "for loops" and so much more

A key component of Julia's ability to combine performance with productivity in a single language is its implementation of dynamic dataflow type inference [15],[11],[**?**]. Unlike type inference algorithms for static languages, this algorithm is tailored to the way dynamic languages work: the typing of code is determined by the flow of data through it. The algorithm works by walking through a program, starting with the types of its input values, and "abstractly interpreting" it: instead of applying the code to values, it applies the code to types, following all branches concurrently and tracking all possible states the program could be in, including all the types each expression could assume.

Since programs can iterate arbitrarily long and recur to arbitrary depths, this process must be iterated until it reaches a fixed point. The design of the type lattice used ensures that the process terminates. Once that point is reached, the program is annotated with upper bounds on the types that each variable and expression can assume.

The dynamic dataflow type inference algorithm allows programs to be automatically annotated with type bounds without forcing the programmer to explicitly specify types. Yet, in dynamic languages it is possible to write programs which inherently cannot be concretely typed. In such cases, dataflow type inference provides what bounds it can, but these may be trivial and useless—i.e. they may not narrow down the set of possible types for an expression at all. However, the design of Julia's programming model and standard library are such that a majority of expressions in typical programs *can* be concretely typed. Moreover, there is a positive correlation between the ability to concretely type code and that code being performance-critical.

This type system supports productive interactions between the programmer and com-

piler. For example, in mathematical environments users often want a square root function defined for reals that returns either a real or complex number depending on the sign of the argument. In Julia this can be defined as

In[22]:
```
# Define xsqrt for real inputs
# return sqrt(complex(x)) if x<0
xsqrt(x::Real) = x < 0 ?  sqrt(complex(x)) :  sqrt(x)
```

This definition works, but will not perform as well as a real-only `sqrt` due to the overhead of tracking whether the result is real or complex. However, in a particular use the programmer might know that a real result is expected, and annotate the call as `xsqrt(y)::Real`. If at runtime, the result is not real, an error is generated.

`Real` is an abstract type, so this code is still generic and works for any real numeric type. (For example, a real integer or a real float or a real rational.) The compiler will combine this declaration with the inferred type of `xsqrt(y)`, recovering exact type information as a result.

A lesson of the numerical computing languages is that one must learn to vectorize to get performance. The mantra is "for loops" are bad, vectorization is good. Indeed one can find the mantra on p.72 of the "1998 Getting Started with Matlab manual" (and other editions):

> Experienced Matlab users like to say "Life is too short to spend writing for loops."

It is not that "for loops" are inherently slow by themselves. The slowness comes from the fact that in the case of most dynamic languages, the system does not have access to the types of the various variables within a loop. Since programs often spend much of their time doing repeated computations, the slowness of a particular operation due to lack of type information is magnified inside a loop. This leads to users often talking about "slow for loops" or "loop overhead".

Consider the example of the cumulative sum earlier. The function definition tells the compiler that the input will be a vector, containing elements of the same type. Within the loop, the compiler is able to infer that the input vector is indexed with scalar integers and the partial sums are stored in the same array as the computation progresses. The sums are computed using the "+" operator. The correct version of `+` is chosen by the system from the 123 `+` methods that are available in Julia at the time of this writing, which define how various types may be added — combinations of integers, floating point numbers, rational numbers, dense arrays, sparse matrices, etc.

```
In[23]: +
```

```
Out[23]: + (generic function with 123 methods)
```

In statically typed languages, full type information is always available at compile time, allowing compilation of a loop into a few machine instructions. This is not the case in most dynamic languages, where the types are discovered at run time, and the cost of determining the types and selecting the right operation can run into hundreds or thousands of instructions.

Julia has a transparent performance model. For example a `Vector{Float64}` as in our example here, always has the same in-memory representation as it would in C or Fortran; one can take a pointer to the first array element and pass it to a C library function using `ccall` and it will just work. The programmer knows exactly how the data is represented and can reason about it. They know that a `Vector{Float64}` does not require any additional heap allocation besides the `Float64` values and that arithmetic operations on these values will be machine arithmetic operations. In the case of say, `Complex128`, Julia stores complex numbers in the same way as C or Fortran. Thus complex arrays are actually arrays of complex values, where the real and imaginary values are stored consecutively. Some systems have taken the path of storing the real and imaginary parts separately, which leads to some convenience for the user, at the cost of performance and interoperability. With the `immutable` keyword, a programmer can also define immutable

data types, and enjoy the same benefits of performance for composite types as for the more primitive number types (bits types). This approach is being used to define many interesting data structures such as small arrays of fixed sizes, which can have much higher performance than the more general array data structure.

The transparency of the C data and performance models has been one of the major reasons for C's long-lived success. One of the design goals of Julia is to have similarly transparent data and performance models. With a sophisticated type system and type inference, Julia achieves both.

6.4 Code Selection

Code selection or code specialization from one point of view is the opposite of code reuse enabled by abstraction. Ironically, viewed another way, it enables abstraction. Julia allows users to overload function names, and select code based on argument types. This can happen at the highest and lowest levels of the software stack. Code specialization lets us optimize for the details of the case at hand. Code abstraction lets calling codes, probably those not yet even written or perhaps not even imagined, work all the way through on structures that may not have been envisioned by the original programmer.

We see this as the ultimate realization of the famous 1908 quip that

> Mathematics is the art of giving the same name to different things.

by noted mathematician Henri Poincaré.[7]

[7] A few versions of this quote are very relevant to Julia's power of abstractions and numerical computing. They are worth pondering:

> It is the harmony of the different parts, their symmetry, and their happy adjustment; it is, in a word, all that introduces order, all that gives them unity, that enables us to obtain a clear comprehension of the whole as well as of the parts. Elegance may result from the feeling of surprise caused by the unlooked-for occurrence of objects not habitually associated. In this, again, it is fruitful, since it discloses thus relations that were until then unrecognized. **Mathematics is the art of giving the same names to different things.**

In this upcoming section we provide examples of how plus can apply to so many objects. Some examples are floating point numbers, or integers. It can also apply to sparse and dense matrices. Another example is the use of the same name, "det", for determinant, for the very different algorithms that apply to very different matrix structures. The use of overloading not only for single argument functions, but for multiple argument functions is already a powerful abstraction.

6.4.1 Dispatch by Argument Type

In Julia one can define a function of two arguments without argument types:

In[24]:
```
g(x,y)=sqrt(x^2+y^2)

# x,y of any type
```

or with argument types. (This is known as optional type annotation.):

In[25]:
```
f(x::Real,    y::Real)     = sqrt(x^2+y^2)
f(x::Complex,y::Complex) = sqrt(abs(x)^2-abs(y)^2)
```

The function $g(x, y)$ will compute $\sqrt{x^2 + y^2}$ no matter what the type of x or y. By contrast, the function $f(x, y)$ amounts to the separation of cases:

---

http://www.nieuwarchief.nl/serie5/pdf/naw5-2012-13-3-154.pdf. and

> One example has just shown us the importance of terms in mathematics; but I could quote many others. It is hardly possible to believe what economy of thought, as Mach used to say, can be effected by a well-chosen term. I think I have already said somewhere that **mathematics is the art of giving the same name to different things**. It is enough that these things, though differing in matter, should be similar in form, to permit of their being, so to speak, run in the same mould. When language has been well chosen, one is astonished to find that all demonstrations made for a known object apply immediately to many new objects: nothing requires to be changed, not even the terms, since the names have become the same.

`http://www-history.mcs.st-andrews.ac.uk/Extras/Poincare_Future.html`

```
if x is Real and y is Real compute sqrt(x^2+y^2}
If x is Complex and y is Complex compute sqrt(abs(x)^2-abs(y)^2)
```

If x and y are not both real or not both complex, then `f(x,y)` is an error.

The notation

In[26]:
```
# Function definitions for the four possibilities of x,y
having Type1/Type2  f(x::Type1, y::Type1) =  code specialized
for both arguments Type1
f(x::Type1, y::Type2) =  code specialized for arguments Type1,
Type2 respectively
f(x::Type2, y::Type1) =  code specialized for arguments Type2,
Type1 respectively
f(x::Type2, y::Type2) =  code specialized for both arguments
Type2
```

is a convenient way of expressing code selection (known as dispatch) based on the types of the first two arguments. Other numerical computing libraries might use Case statements, or If/ElseIf constructs. Julia avoids the time that can be wasted during this code selection process which degrades performance.

In Julia, the dispatch by type goes straight to the compiler leading to remarkable efficiencies. These efficiencies can be measured by small numbers of lines of assembler or by short execution times.

We have not seen this in the literature but it seems worthwhile to point out four possibilities:

1. static single dispatch (not done)

2. static multiple dispatch (frequent in static languages, e.g. C++ overloading)

3. dynamic single dispatch (MATLAB's object oriented system might fall in this category though it has its own special characteristics)

4. dynamic multiple dispatch (usually just called multiple dispatch).

In Section 6.4.3 we discuss the comparison with traditional object oriented approaches. Class-based object oriented programming could reasonably be called dynamic single dispatch, and overloading could reasonably be called static multiple dispatch. Julia's (dynamic!) multiple dispatch approach is more flexible and adaptable while still retaining powerful performance capabilities. Julia programmers often find that dynamic multiple dispatch makes it easier to structure their programs in ways that are closer to the underlying science.

6.4.2 Code selection from bits to matrices

Julia uses the same mechanism for code selection at all levels, from the top to the bottom. In Sections 6.4.2, 6.4.2, and 6.4.2 we consider code selection problems that have the common general format defined in In[24] above.

f	Function	Operand Types
Low Level "+"	Add Numbers	{Float , Int}
High Level "+"	Add Matrices	{Dense Matrix , Sparse Matrix}
" * "	Scale or Compose	{Function , Number }

Summing Numbers: Floats and Ints

We begin at the lowest level. Mathematically, integers are thought of as being special real numbers, but on a computer, an Int and a Float have two very different representations. Ignoring for a moment that there are even many choices of Int and Float representations, if we add two numbers, code selection based on numerical representation is taking place at a very low level. Most users are blissfully unaware of this code selection, because it is hidden

somewhere that is usually off-limits to the user. Nonetheless, one can follow the evolution of the high level code all the way down to the assembler level which ultimately would reveal an ADD instruction for integer addition, and, for example, the AVX[8] instruction VADDSD[9] for floating point addition in the language of x86 assembly level instructions. The point being these are ultimately two different algorithms being called, one for a pair of Ints and one for a pair of Floats.

Figure **??** takes a close look at what a computer must do to perform `x+y` depending on whether (x,y) is (Int,Int), (Float,Float), or (Int,Float) respectively. In the first case, an integer add is called, while in the second case a float add is called. In the last case, a promotion of the int to float is called through the x86 instruction VCVTSI2SD[10], and then the float add follows.

It is instructive to build a Julia simulator in Julia itself. Let us define the aforemen-

[8] AVX: Adanced Vector eXtension to the x86 instruction set

[9] VADDSD: Vector ADD Scalar Double-precision

[10] VCVTSI2SD: Vector ConVerT Doubleword (Scalar) Integer to(2) Scalar Double Precision Floating-Point Value

tioned assembler instructions using Julia.

In[27]:
```
# Simulate the assembly level add, vaddsd, and vcvtsi2sd
commands

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

In[28]:
```
# Simulate Julia's definition of + using ⊕
# To type ⊕, type as in TeX, \oplus and hit the <tab> key
⊕(x::Int,    y::Int)     = add(x,y)
⊕(x::Float64,y::Float64) = vaddsd(x,y)
⊕(x::Int,    y::Float64) = vaddsd(vcvtsi2sd(x),y)
⊕(x::Float64,y::Int)     = y ⊕ x
```

In[29]:
```
methods(⊕)
```

Out[29]:
```
4 methods for generic function ⊕:
⊕ (x::Int64,y::Int64) at In[26]:3
⊕ (x::Float64,y::Float64) at In[26]:4
⊕ (x::Int64,y::Float64) at In[26]:5
⊕ (x::Float64,y::Int64) at In[26]:6
```

Summing Matrices: Dense and Sparse

We now move to a much higher level: matrix addition. The versatile "+" symbol lets us add matrices. Mathematically, sparse matrices are thought of as being special matrices with enough zero entries. On a computer, dense matrices are (usually) contiguous blocks of data with a few parameters attached, while sparse matrices (which may be stored

in many ways) require storage of index information one way or another. If we add two matrices, code selection must take place depending on whether the summands are (dense,dense), (dense,sparse), (sparse,dense) or (sparse,sparse).

While this is at a much higher level, the basic pattern is unmistakably the same as that of Section 6.4.2. While including all the bells and whistles is not appropriate in this paper, we can show one way of implementing and algorithms that works with dense matrices if either A or B are dense, but uses a sparse algorithm if both are sparse.

In[34]:
```
# Dense + Dense
⊕(A::Matrix, B::Matrix) =
     [A[i,j]+B[i,j] for i in 1:size(A,1),j in 1:size(A,2)]
# Dense + Sparse
⊕(A::Matrix, B::AbstractSparseMatrix) = A ⊕ full(B)
# Sparse + Dense
⊕(A::AbstractSparseMatrix,B::Matrix)  = B ⊕ A
# Sparse + Sparse is best written using the long form
function definition:  function ⊕(A::AbstractSparseMatrix,
B::AbstractSparseMatrix)
    C=copy(A)
    (i,j)=findn(B)
    for k=1:length(i)
        C[i[k],j[k]]+=B[i[k],j[k]]
    end
    C
end
```

We now have eight methods for $\oplus$, four for the low level sum, and four more for the high level sum. The most important point is that the mechanism for code selection, this

dispatch notation which goes straight to the compiler, is the same.

```
In[35]: methods(⊕)
```

```
Out[35]: 8 methods for generic function ⊕:
```

a listing of all eight follows: four low level, four matrix level

Further examples

We have a few further examples of the framework provided by In[24] at the beginning of this section of the paper.

It is possible to model mathematical notations that are often used in print, but are difficult to employ in programs. For example, we can teach the computer some natural ways to multiply numbers and functions. Suppose that a and t are scalars, and f and g are functions, and we wish to define

1. **Number x Function = scale output:** $a * g$ is the function that takes x to $a * g(x)$
2. **Function x Number = scale argument :** $f * t$ is the function that takes x to $f(tx)$ and
3. **Function x Function = composition of functions:** $f * g$ is the function that takes x to $f(g(x))$.

If you are a mathematician who does not program, you would not see the fuss. If you thought how you might implement this in your favorite computer language, you might immediately see the benefit. In Julia, multiple dispatch makes all three uses of * easy to express:

In[36]:
```
*(a::Number, g::Function)= x->a*g(x)    # Scale output
*(f::Function,t::Number) = x->f(t*x)    # Scale argument
*(f::Function,g::Function)= x->f(g(x)) # Function
composition
```

Here, multiplication is dispatched by the type of its first and second arguments. It goes the usual way if both are numbers, but there are three new ways if one, the other, or both are functions.

These definitions exist as part of a larger system of generic definitions, which can be reused by later definitions. Consider the case of the mathematician Gauss' preference for $\sin^2\phi$ to refer to $\sin(\sin(\phi))$ and not $\sin(\phi)^2$ (writing "$\sin^2(\phi)$ is odious to me, even though Laplace made use of it."(Figure **??**).) By defining `*(f::Function,g::Function)= x->f(g(x))`, `(f^2)(x)` automatically computes $f(f(x))$ as Gauss wanted. This is a consequence of a generic definition that evaluates `x^2` as `x*x` no matter how `x*x` is defined.

This paradigm is a natural fit for numerical computing, since so many important operations involve interactions among multiple values or entities. Binary arithmetic operators are obvious examples, but other uses abound. For example,[11] code performing collision detection among different shapes can define

```
In[37]: function collide(me::Circle, other::Rectangle)
        function collide(me::Polygon, other::Circle)
        function collide(me::Polygon, other::Rectangle)
```

When the call `collide(self, other)` takes place, the appropriate method is selected based on both arguments. This dispatch is dynamic, based on run-time types, unlike function overloading in C++. This design point is important, since it provides full flexibility — the expected method is called no matter how much run-time variation there is in the types of shape arguments, and independent of whether the compiler can prove anything about those types. Dynamic multiple dispatch encompasses both object-oriented techniques used in general-purpose programming, and the behaviors of mathematical objects compilers do not know how to reason about.

Note this example allows for three shape types: Circle, Rectangle, Polygon. There could thus be nine possible functions total of a function with two arguments and three types for each argument.

[11] From `http://assoc.tumblr.com/post/71454527084/cool-things-you-can-do-in-julia`.

6.4.3 Is "code selection" just traditional object oriented programming?

It is worth crystallizing some key aspects of the code selection as described above.

1. The same name can be used for different functions. (See Footnote 5). (Method or function overloading.)

2. The collection of functions that might be called by the same name is thought of as an entity itself. (Generic functions.)

3. Code selection at the lowest and highest levels use one and the same mechanism

4. Which method is called is based entirely on the types of the arguments of the methods. (This is sometimes the emphasis of the term ad-hoc polymorphism.)

5. The types of the arguments of a method that is being called may or may not be knowable by a static compiler. Instead, the method may be chosen dynamically at runtime as the types become known. (Dynamic Dispatch) (See Figure **??**.) In particular, one can recover from a loss of type information.

6. The method is not chosen by only one argument (Single Dispatch) but rather by all the arguments (Multiple Dispatch)

7. Julia is not encumbered by the encapsulation restrictions (class based methods) of most object oriented languages. The generic functions play a more important role than the types. (Some call this "verb" based languages as opposed to most object oriented languages being "noun" based. In numerical computing, it is the concept of "solve $Ax = b$" that often feels more primary, at the highest level, rather than whether the matrix A is full, sparse, or structured.)

Readers familiar with Java might think, so what? One can easily create methods based on the types of the arguments. An example is provided in Figure **??**.

However a moment's thought shows that the following dynamic situation in Julia is impossible to express in Java:

In[38]:
```
# It is possible for a static compiler to know that x,y
are Float
x = randbool() ?  1.0 :  2.0
y = randbool() ?  1.0 :  2.0
x+y

# It is impossible to know until runtime if x,y are Int or
Float
x = randbool() ?  1 :  1.0
y = randbool() ?  1 :  1.0
x+y
```

In Julia as in mathematics, functions are as important as their arguments. Perhaps even more so. We saw in Out[21] that "+" already has 123 methods attached, while Out[8] and Out[28] are functions with one method. In Out[33] we took advantage of Julia's ability to name variables and methods with unicode to create the generic $\oplus$ with eight methods. We can create a new function `foo` and gave it six definitions depending on the combination of types. In the following example we introduce terms from computer science language research for the benefit of an audience of numerical computing

programmers.

In[39]:
```
# Define a generic function with 6 methods.  Each method
is itself a
# function.  In Julia generic functions are far more
convenient than the
# multitude of case statements seen in other languages.
When Julia sees
# foo, it decides which method to use, rather than first
seeing and deciding
# based on the type.

foo() = "Empty input"
foo(x::Int) = x
foo(S::String) = length(S)
foo(x::Int, S::String) = "An Int and a String"
foo(x::Float64,y::Float64) = sqrt(x^2+y^2)
foo(a::Any,b::String)= "Something more general than an Int and a String"

# The function name foo is overloaded.  This is an example
of polymorphism.
# In the jargon of computer languages this is called
ad-hoc polymorphism.
# The multiple dynamic dispatch idea captures the notion
that the generic
# function is deciphered dynamically at runtime.  One of
the six choices
# will be made or an error will occur.
```

Out[39]: foo (generic function with 6 methods)

Any one instance of `foo` is known as a method or function. The collection of six methods is referred to as a **generic function**. Contemplating the Poincaré quote in Footnote 5, it is handy to reason about everything that you are giving the same name. In real life coding, one tends to use the same name when the abstraction makes a great deal of sense. Humans often lose sight it is an abstraction. That we use "+" for ints,floats, dense matrices, and sparse matrices is the same name for different things. Methods are grouped into (generic) functions. A generic function can be applied to several arguments, and the method with the most specific signature matching the arguments is invoked.

Readers familiar with MATLAB may be familiar with MATLAB's single dispatch mechanism. It is unusual in that it is not completely class based, as the code selection is based on MATLAB's own custom hierarchy. In MATLAB the leftmost object has precedence, but user-defined classes have precedence over built-in classes. MATLAB also has a mechanism to create a custom hierarchy.

Julia generally shuns the notion of "built-in" vs. "user-defined" preferring to focus on the method to be performed based on the combination of types, and obtaining high performance as a byproduct. A high level library writer, which we do not distinguish from any user, has to match the best algorithm for the best input structure. A sparse matrix would match to a sparse routine, a dense matrix to a dense routine. A low level language designer has to make sure that integers are added with an integer adder, and floating points are added with a float adder. Despite the very different levels, the reader might recognize that deep down, these are both examples of code being selected to match the structure of the problem.

Readers familiar with object-oriented paradigms such as C++ or Java are most likely familiar with the approach of encapsulating methods inside classes. This is very similar to our single dispatch examples, where we have a `newdet` method for each type of matrix. Julia's more general multiple dispatch mechanism (also known as generic functions, or multi-methods) is a paradigm where methods are defined on combinations of data types

(classes) Julia has proven that this is remarkably well suited for numerical computing.

A class based language might express the sum of a sparse matrix with a full matrix as follows: `A_sparse_matrix.plus(A_full_matrix)`. Similarly it might express indexing as

`A_sparse_matrix.sub(A_full_matrix)` . If a tridiagonal were added to the system, one has to find the method `plus` or `subsref` which is encapsulated in the sparse matrix class, modify it and test it. Similarly, one has to modify every full matrix method, etc. We believe that class-based methods, which can be taken quite far, are not sufficiently powerful to express the full gamut of abstractions in scientific computing. Further, the burdens of encapsulation create a wall around objects and methods that are counterproductive for numerical computing.

The generic function idea captures the notion that a method for a general operation on pairs of matrices may exist (e.g. "+") but if a more specific operation is possible (e.g. "+" on sparse matrices, or "+" on a special matrix structure like Bidiagonal), then the more specific operation is used. We also mention indexing as another example, Why should the indexee take precedence over the index?

Quantifying use of multiple dispatch

In [**?**] we performed an analysis to substantiate the claim that multiple dispatch, an esoteric idea for numerical computing from computer languages, finds its killer application in scientific computing. We wanted to answer for ourselves the question of whether there was really anything different about how Julia uses multiple dispatch.

Table **??** gives an answer in terms of Dispatch ratio (DR), Choice ratio (CR). and Degree of specialization (DoS). While multiple dispatch is an idea that has been circulating for some time, its application to numerical computing appears to have significantly favorable characteristics compared to previous applications.

6.4.4 Case Study for Numerical Computing

For a reader wishing to get started with some of these powerful features in Julia we provide an introductory example.

Determinant: Simple Single Dispatch

In traditional numerical computing there were people with special skills known as library writers. Most users were, well, just users of libraries. In this case study, we show how anybody can dispatch a new determinant function based solely on the type of the argument.

For triangular and diagonal structures the obvious formulas are used. For general matrices, the programmer will compute QR and use the diagonal elements of R.[12] For symmetric tridiagonals the usual 3-term recurrence formula is used. (The first four are

[12] LU is more efficient. We simply wanted to illustrate other ways are possible.

defined as one line functions; the symmetric tridiagonal uses the long form.)

In[40]:
```
# Simple determinants defined using the short form for
functions
newdet(x::Number) = x
newdet(A::Diagonal ) = prod(diag(A))
newdet(A::Triangular) = prod(diag(A))
newdet(A::Matrix) = -prod(diag(qrfact(full(A))[:R]))*(-1)^size(A,1)

# Tridiagonal determinant defined using the long form for
functions
function newdet(A::SymTridiagonal)
    # Assign c and d as a pair
    c,d = 1, A[1,1]
    for i=2:size(A,1)
        # temp=d, d=the expression, c=temp
        c,d = d, d*A[i,i]-c*A[i,i-1]^2
    end
    d
end
```

We have illustrated a mechanism to select a determinant formula at runtime based on the type of the input argument. If Julia knows an argument type early, it can make use of this information for performance. If it does not, code selection can still happen, at runtime. The reason why Julia can still perform well is that once code selection based on type occurs, Julia can return to performing well once insider the method.

A Symmetric Arrow Matrix Type

In the field of Matrix Computations, there are matrix structures and operations on these matrices. In Julia, these structures exist as Julia types. Julia has a number of predefined matrix structure types: (dense) `Matrix`, (compressed sparse column) `SparseMatrixCSC`, `Symmetric`, `Hermitian`, `SymTridiagonal`, `Bidiagonal`, `Tridiagonal`. `Diagonal`, and `Triangular` are all examples of Julia's matrix structures.

The operations on these matrices exist as Julia's methods. Familiar examples of operations are indexing, determinant, size, and matrix addition. Since matrix addition takes two arguments, it may be necessary to reconcile two different types when computing the sum.

Some languages do not allow you to extend their built in methods. This ability is known as external dispatch. In the following example, we illustrate how the user can add symmetric arrow matrices to the system, and then add a specialized `det` method to compute the determinant of a symmetric arrow matrix efficiently.

In[41]:
```
# Define a Symmetric Arrow Matrix Type
immutable SymArrow{T} <:  AbstractMatrix{T}
    dv::Vector{T} # diagonal
    ev::Vector{T} # 1st row[2:n]
  end
```

In[42]:
```
# Define its size
importall Base
size(A::SymArrow, dim::Integer) = size(A.dv,1)
size(A::SymArrow)= size(A,1), size(A,1)
```

Out[42]: `size (generic function with 52 methods)`

In[43]:
```
# Index into a SymArrow
function getindex(A::SymArrow,i::Integer,j::Integer)
    if i==j; return A.dv[i]
      elseif i==1; return A.ev[j-1]
      elseif j==1; return A.ev[i-1]
      else return zero(typeof(A.dv[1]))
    end
end
```

Out[43]:
```
getindex (generic function with 168 methods)
```

In[44]:
```
# Dense version of SymArrow
full(A::SymArrow) =[A[i,j] for i=1:size(A,1),
j=1:size(A,2)]
```

Out[44]:
```
full (generic function with 17 methods)
```

In[45]:
```
# An example
S=SymArrow([1,2,3,4,5],[6,7,8,9])
```

Out[45]:
```
5x5 SymArrow{Int64}:
1 6 7 8 9
6 2 0 0 0
7 0 3 0 0
8 0 0 4 0
9 0 0 0 5
```

In[46]:
```
# det for SymArrow (external dispatch example)
function exc_prod(v) # prod(v)/v[i]
    [prod(v[[1:(i-1),(i+1):end]]) for i=1:size(v,1)]
end
# det for SymArrow formula
det(A::SymArrow) = prod(A.dv)-sum(A.ev.^2.*exc_prod(A.dv[2:end]))
```

Out[46]: `det (generic function with 17 methods)`

The above julia code uses the special formula

$$\det(A) = \prod_{i=1}^{n} d_i - \sum_{i=2}^{n} e_i^2 \prod_{2 \le j \ne i \le n} d_j,$$

valid for symmetric arrow matrices with diagonal d and first row starting with the second entry e.

In Julia terminology `det` and `newdet` are *functions.* In some numerical computing languages, a function might begin with a lot of argument checking to pick which algorithm to use. In Julia, one creates a number of *methods.* Thus `newdet` on a diagonal is one method for `newdet`, and `newdet` on a triangular matrix is a second method. `det` on a `SymArrow` is a new method for `det`. Code is selected, in advance if the compiler knows the type, otherwise the code is selected at run time. The selection of code is known as *dispatch.*

We have seen a number of examples of code selection for single dispatch. We can now turn to a very powerful feature, Julia's multiple dispatch mechanism. Now that we have created a symmetric arrow matrix, we might want to add it to all possible matrices of all types. However, we might notice that a symmetric arrow plus a diagonal does not require operations on full dense matrices.

The code below starts with the most general case, and then allows for specialization

for the symmetric arrow and diagonal sum:

In[47]:
```
# SymArrow + Any Matrix:  (Fallback:  add full dense
arrays )
+(A::SymArrow, B::Matrix) = full(A)+B
+(B::Matrix, A::SymArrow) = A+B
# SymArrow + Diagonal:  (Special case:  add diagonals,
copy off-diagonal)
+(A::SymArrow, B::Diagonal) = SymArrow(A.dv+B.diag,A.ev)
+(B::Diagonal, A::SymArrow) = A+B
```

6.5 Code reuse: Code Generation is not only for the compiler

At the crossroads of code selection and code reuse is the linear algebra software engineer's dilemma. The complexity of the full solution has been nicely captured in the context of LAPACK and ScaLAPACK by Demmel and Dongarra, et.al., [?] and reproduced verbatim here:

```
(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)
```

Obviously (1) and (6) and perhaps (4) and even (5) are about code selection. (2) and (3) are about code reuse at the high level, and code selection at the lowest levels.

In the language of Computer Science, code reuse is about taking advantage of polymorphism. In the general language of mathematics it's about taking advantage of abstraction, or the sameness of two things. Either way, programs are efficient, powerful, and maintainable if programmers are given powerful mechanisms to reuse code.

Reusing code can be very tricky. Much of the development of modern programming language theory, both static and dynamic, can be understood by following the emergence

of various forms of polymorphism. Saying the same notion another way, figuring out how to enable users to reuse code is perhaps far more difficult than training mathematicians to recognize that two different mathematical objects share sufficient commonality that they may be given the same name.

6.5.1 Example: Non-floating point linear algebra

Increasingly, the applicability of linear algebra has gone well beyond the LAPACK world of floating point numbers. These days linear algebra is being performed on, say, high precision numbers, integers, elements of finite fields, or rational numbers.

There will always be a special place for the BLAS, and the performance it provides for floating point numbers. Nonetheless, linear algebra operations like "inv" transcend any one data type. One can write a general "inv" and as long as the necessary operations are available, the code just works. That is the power of code reuse.

Here we show an example of rational types just working. A rational could be replaced by other constructs, which if they have well defined operations of algebra, will continue to work.

In[48]:
```
nHilbert = 8
```

Out[48]:
```
8
```

In[49]:
```
H = Rational{BigInt}[1//(i+j-1) for i=1:nHilbert,
j=1:nHilbert]
```

```
Out[49]: 8x8 Array{Rational{BigInt},2}:

 1//1  1//2  1//3   1//4   1//5   1//6   1//7   1//8
 1//2  1//3  1//4   1//5   1//6   1//7   1//8   1//9
 1//3  1//4  1//5   1//6   1//7   1//8   1//9   1//10
 1//4  1//5  1//6   1//7   1//8   1//9   1//10  1//11
 1//5  1//6  1//7   1//8   1//9   1//10  1//11  1//12
 1//6  1//7  1//8   1//9   1//10  1//11  1//12  1//13
 1//7  1//8  1//9   1//10  1//11  1//12  1//13  1//14
 1//8  1//9  1//10  1//11  1//12  1//13  1//14  1//15
```

```
In[50]: inv(H)
```

```
Out[50]: 8x8 Array{Rational{BigInt},2}:

     64//1      -2016//1  ...  -288288//1       192192//1      -51480//1
  -2016//1       84672//1        15567552//1   -10594584//1     2882880//1
  20160//1     -952560//1      -204324120//1    141261120//1   -38918880//1
 -92400//1     4656960//1       1109908800//1  -776936160//1    216216000//1
 221760//1   -11642400//1      -2996753760//1   2118916800//1  -594594000//1
-288288//1    15567552//1  ...   4249941696//1  -3030051024//1    856215360//1
 192192//1   -10594584//1      -3030051024//1   2175421248//1  -618377760//1
 -51480//1     2882880//1        856215360//1  -618377760//1    176679360//1
```

6.5.2 Generic Programming for library development

Julia blurs the line between library developer and user. The modern numerical computing world depends heavily on domain libraries for all areas of applied math, and specialized analyses for different branches of science. These libraries make end users more productive, but developing the libraries themselves is a difficult problem. A useful library needs to be

both high performance and generic, providing enough flexibility to adapt to the varying problems of different users.

Meeting these requirements places a large burden on library developers. The need for code that is both generic and performant pushes programming languages to their limits, requiring use of advanced features such as C++ templates. Next, the popularity of dynamic languages means that interfacing work is needed to make these libraries callable from Python, R, and other systems. Overall, this workflow requires not only domain knowledge, but a high level of programming skill and knowledge of language internals. These requirements on time and expertise are unreasonable.

The design of Julia changes this situation. While being "high level" and "fast" are prerequisites to solving the problem, the true requirements involve subtleties that are not addressed just by taking any high level language and speeding it up. To obtain the best performance for complex library constructs, a compiler must understand code more deeply, and perform more computations at compile time. This is where a fresh language design is needed.

A good example is determining the ranks of multi-dimensional arrays. Numerical computing systems typically have complex rules for how various functions operate on array dimensions, and knowing the ranks of arrays at compile time is crucial for performance. Therefore compilers for "array languages" like APL and MATLAB typically have built-in rules for such functions. However built-in rules do not address the flexibility needs of library developers. More general mechanisms like C++ templates are needed to "teach" compilers new rules. Unfortunately, templates are notoriously difficult to use.

We addressed the array-rank-inference problem in a recent paper [**?**]. It turns out that Julia's combination of multiple dispatch and dataflow type inference allows array rank rules to be defined in a natural way, with minimal code. Furthermore, this style of expression allows our compiler to infer array ranks without specialized array knowledge being built in. The following example of this technique is taken directly from Julia's

standard library:

In[51]:
```
index_shape(I::Real...)  = ()
index_shape(i, I...)  = tuple(length(i),
index_shape(I...)...)
```

The arguments are indices that might be used to index an array, and the output is the shape. These two lines define a generic function that computes the shape of the array I resulting from an indexing operation. It implements the rule that trailing dimensions indexed with scalars are dropped.

This code may look cryptic to the newcomer, so let's break it down.

In[52]:
```
# Indexing entirely by scalars returns an empty tuple
index_shape(I::Real...)  = ()
```

The `I::Real` indicates a real scalar number type. The `...` indicates a sequence of such types, which can be any length including 0. Thus calling `index_shape` with any sequence of scalars yields an empty array:

In[53]:
```
println( index_shape() )
println( index_shape(1,2,3) )
println( index_shape(1,2,3,4) )
```

Out[53]:
```
()
()
()
```

The method on the second line is dispatched if any one argument is not a scalar:

In[54]:
```
# Peel off the length of the first argument, and dispatch
the remaining part
index_shape(i,I...)=tuple(length(i),index_shape(I...)...)
```

In[55]:
```
println( index_shape(7:9) )
println( index_shape(5,7:9) )
println( index_shape(7:9,1,2,3,4) )
```

Out[55]:
```
(3,)
(1,3)
(3,)
```

Different rules (e.g. dropping all dimensions indexed with scalars) can be obtained by adjusting the definitions slightly. When dataflow type inference is applied to these definitions for any particular invocation, a tuple type of the correct length emerges, telling the compiler the array rank.

We emphasize that this can be written in other ways in other numerical computing languages, but doing it this way gives the information to the compiler. This is the moral of Julia, that the programmer can help provide information to the compiler so that code is not slowed down at run time.

6.5.3 Macros

Julia has a macro system that provides easy custom code generation, bringing a level of performance that is otherwise difficult to achieve. A macro is a function that runs at parse-time, and takes parsed symbolic expressions in and returns transformed symbolic expressions out, which are inserted into the code for later compilation.

For example, a library developer implemented an `@evalpoly` macro that uses Horner's

rule to evaluate polynomials efficiently. Consider

```
In[56]: @evalpoly(10,3,4,5,6)
```

which returns 6543 (the polynomial $3 + 4x + 5x + 6x^2$, evaluated at 10 with Horner's rule). Julia allows us to see the inline generated code with the command

```
In[57]: macroexpand(:@evalpoly(10,3,4,5,6))
```

We reproduce the key lines below

```
Out[57]: #471#t = 10 # Store 10 into a variable named #471#t
         Base.Math.+(3,Base.Math.*(#471#t,Base.Math.+(4,Base.Math.*
         (#471#t,Base.Math.+(5,Base.Math.*(#471#t,6)))) ))
```

This code-generating macro only needs to produce the correct symbolic structure, and Julia's compiler handles the remaining details of fast native code generation. Since polynomial evaluation is so important for numerical library software it is critical that users can evaluate polynomials as fast as possible. The overhead of implementing an explicit loop, accessing coefficients in an array, and possibly a subroutine call (if it is not inlined), is substantial compared to just inlining the whole polynomial evaluation.

Steven Johnson reports in his EuroSciPy https://www.euroscipy.org/2014 notebook https://github.com/stevengj/Julia-EuroSciPy14/blob/master/Metaprogramming.ipynb

> This is precisely how erfinv is implemented in Julia (in single and double precision), and is 3 to 4 faster than the compiled (Fortran?) code in Matlab, and 2 to 3 faster than the compiled (Fortran Cephes) code used in SciPy.
>
> The difference (at least in Cephes) seems to be mainly that they have explicit arrays of polynomial coefficients and call a subroutine for Horner's rule, versus inlining it via a macro.

Johnson also used the same trick in his implementation of the digamma special func-

tion for complex argumens [13] following an idea of Knuth:

> As described in Knuth TAOCP vol. 2, sec. 4.6.4, there is actually an algorithm even better than Horner's rule for evaluating polynomials p(z) at complex arguments (but with real coefficients): you can save almost a factor of two for high degrees. It is so complicated that it is basically only usable via code generation, so it would be especially nice to modify the @horner macro to switch to this for complex arguments.

No sooner than this was proposed, the macro was rewritten to allow for this case giving a factor of four performance improvement on all real polynomials evaluated at complex arguments.

6.5.4 Just-In-Time compilation and code specialization

.

Assuming that users want performance, Julia specializes code for run-time types by default. This technique is fairly effective, often yielding performance within a factor of 2 of C (Fig. **??**).

Through specialization, our compiler can remove overhead for user defined types with sophisticated behaviors. The Units package written by Keno Fischer[14] is a good example. With this package installed, it is possible to ask for the sum of two lengths:

```
In[58]: Pkg.add("SIUnits") # Needed once to download the package
        using SIUnits
        1Meter + 2Meter
```

```
Out[58]: 3m
```

[13] https://github.com/JuliaLang/julia/issues/7033
[14] https://github.com/Keno/SIUnits.jl

The + methods defined in the library get compiled to fast machine code, consisting only of an `add` instruction and stack operations, with the `Meter`s completely optimized away. It is possible to see the x86 assembler code any time with the `@code_native` command.

(As already mentioned in the caption to Figure **??**, there are many online listings such as

`http://docs.oracle.com/cd/E36784_01/pdf/E36859.pdf` or

`http://en.wikipedia.org/wiki/X86_instruction_listings` for users to learn what all the commands are doing. Julia users who never looked at assembler before, can get to understand assembler very quickly by simply trying out a few simple commands.)

```
In[59]: @code_native 1Meter + 2Meter
```

```
Out[59]:     .section __TEXT,__text,regular,pure_instructions
            Filename:  /Users/viral/.julia/SIUnits/src/SIUnits.jl
            Source line:  122
              push RBP
              mov RBP, RSP
            Source line:  122
              add RDI, RSI
            Source line:  123
              mov RAX, RDI
              pop RBP
              ret
```

The data is stored exactly as an array of floating-point values with no overhead for storage or computation, because the units are attached to the type of the array rather than to the elements. The trained eye sees this as minimal code generated for the instruction. Specifically there is some register activity and simply one add instruction (highlighted here).

6.6 Language and standard library design

Seemingly innocuous design choices in a language can have profound, pervasive performance implications. These are often overlooked in languages that were not designed from the beginning to be able to deliver excellent performance. Other aspects of language and library design affect the usability, composability, and power of the provided functionality.

6.6.1 Integer arithmetic

A simple but crucial example of a performance-critical language design choice is integer arithmetic. Julia uses machine arithmetic for integer computations. This means that the range of Int values is bounded and wraps around at either end so that adding, subtracting and multiplying integers can overflow or underflow, leading to results that are familiar to C and Fortran programmers but which can be unsettling to users of systems like Mathematica or Python:

```
In[60]: typemax(Int)
Out[60]: 9223372036854775807

In[61]: ans+1
Out[61]: -9223372036854775808

In[62]: -ans
Out[62]: -9223372036854775808

In[63]: 2*ans
Out[63]: 0
```

In both Mathematica and Python, integers behave like mathematical integers, growing forever larger or smaller, never overflowing. While this may be an acceptable choice if

only end-users will ever write code with the system's integers, it becomes rapidly clear that it is too slow if every loop in every function uses these integers to count its loops and compute indices and offsets.

Since machine multiplication and addition are associative and distribute, Julia is free to aggressively optimize simple little functions like `f(k) = 5k-1`. The machine code for this function is just this:

```
In[64]: code_native(f,(Int,))
```

```
Out[64]:    .section __TEXT,__text,regular,pure_instructions
          Filename: none
          Source line: 1
            push RBP
            mov RBP, RSP
          Source line: 1
            lea RAX, QWORD PTR [RDI + 4*RDI - 1]
            pop RBP
            ret
```

The actual body of the function is a single `lea` instruction, which computes the integer multiply and add at once. (Assembler names such as "'Load Effective Address," have grown irrelevant and one may think of LEA as simply an integer operation.) The benefit of minimal code generation is even more beneficial when `f` gets inlined into the inner

loop of another function:

In[65]:
```
function g(k,n)
    for i = 1:n
      k = f(k)
    end
    k
end
```

Out[65]: g (generic function with 2 methods)

In[66]:
```
code_native(g,(Int,Int))
```

```
Out[66]:    .section __TEXT,__text,regular,pure_instructions
          Filename:  none
          Source line:  3
            push RBP
            mov RBP, RSP
            test RSI, RSI
            jle 22
            mov EAX, 1
          Source line:  3
            lea RDI, QWORD PTR [RDI + 4*RDI - 1]
            inc RAX
            cmp RAX, RSI
          Source line:  2
            jle -17
          Source line:  5
            mov RAX, RDI
            pop RBP
          ret
```

Since the call to `f` gets inlined, the loop body ends up being just a single `lea` instruc-

tion. Next, consider what happens if we make the number of loop iterations fixed:

In[67]:
```
# 10 Iterations of f(k)=5k-1 on integers
function g(k)
    for i = 1:10
      k = f(k)
    end
    k
end
```

Out[67]:
```
g (generic function with 2 methods)
```

In[68]:
```
code_native(g,(Int,))
```

Out[68]:
```
    .section __TEXT,__text,regular,pure_instructions
Filename:  none
Source line:  3
  push RBP
  mov RBP, RSP
Source line:  3
  imul RAX, RDI, 9765625
  add RAX, -2441406
Source line:  5
  pop RBP
  ret
```

Because the compiler knows that integer addition and multiplication are associative and that multiplication distributes over addition it can optimize the entire loop down to just a multiply and an add. Indeed, if $f(k) = 5k - 1$, it is true that the tenfold iterate $f^{(10)}(k) = -2441406 + 9765625k$.

6.6.2 A powerful approach to linear algebra

Matrix factorizations

For decades, orthogonal matrices have been represented internally as products of Householder matrices stored in terms of vectors, and displayed for humans as matrix elements. LU factorizations are often performed in place, storing the L and U information together in the data locations originally occupied by A. All this speaks to the fact that matrix factorizations deserve to be first class objects in a linear algebra system.

In Julia, thanks to the contributions of Andreas Noack Jensen and many others, these structures are indeed first class objects. The structure `QRCompactWY` holds a compact Q and an R in memory. Similarly an `LU` holds an L and U in packed form in memory. Through the magic of multiple dispatch, we can solve linear systems, extract the pieces, and do least squares directly on these structures.

The QR example is even more fascinating. Suppose one computes QR of a 5×3 matrix. What is the size of Q? The right answer, of course, is that it depends: it could be 5×5 or 5×3. The underlying representation is the same.

In Julia one can compute `Aqr = qrfact(rand(5,3))`. Then one can take `Q=Aqr[:Q]`. This Q retains its clever underlying structure and therefore is efficient and applicable when multiplying vectors of length 5 or length 3, contrary to the rules of freshman linear algebra, but welcome in numerical libraries for saving space and faster computations.

In[69]:
```
Aqr = qrfact(rand(5,3));
Q = Aqr[:Q]
```

```
Out[69]: 5x5 QRCompactWYQ{Float64}:
          -0.251536   0.154937   0.77616
          -0.101711  -0.608041   0.512436
          -0.755597   0.217326  -0.0987686
          -0.014734  -0.714761  -0.210326
          -0.596021  -0.219468  -0.284593

In[70]: Q*rand(5)
Out[70]: 5-element Array{Float64,1}:
           0.755784
          -1.10069
          -0.757399
          -0.272477
          -0.0510777
In[71]: Q*rand(3)
Out[71]: 5-element Array{Float64,1}:
           0.152596
          -0.0562855
          -0.569295
          -0.303167
          -0.644554
```

User-extensible wrappers for BLAS and LAPACK

The tradition in linear algebra is to leave the coding to LAPACK writers, and call LAPACK for speed and accuracy. This has worked fairly well, but Julia exposes considerable opportunities for improvement.

Firstly, all of LAPACK is available to Julia users, not just the most common func-

tions. All LAPACK wrappers are implemented fully in Julia code, using "ccall"[15], which does not require a C compiler, and can be called directly from the interactive Julia prompt. This makes it easy for users to contribute LAPACK functionality, and that is how Julia's LAPACK functionality has grown bit by bit. Wrappers for missing LAPACK functionality can also be added by users in their own code.

Consider the following example that implements the Cholesky factorization by calling LAPACK's `xPOTRF`. It uses Julia's metaprogramming facilities to generate four functions, each corresponding to the `xPOTRF` functions for `Float32`, `Float64`, `Complex64`, and `Complex128` types. The actual call to the Fortran functions is wrapped in `ccall`. Finally, the `chol` function provides a user-accessible way to compute the factorization.

[15] `http://docs.julialang.org/en/latest/manual/calling-c-and-fortran-code/`

It is easy to modify the template below for any LAPACK call.

In[72]:

```
# Generate calls to LAPACK's Cholesky for double, single,
etc.
# xPOTRF refers to POsitive definite TRiangular Factor
# LAPACK signature:  SUBROUTINE DPOTRF( UPLO, N, A, LDA,
INFO )
# LAPACK documentation:

*  UPLO    (input) CHARACTER*1
*          = 'U':  Upper triangle of A is stored;
*          = 'L':  Lower triangle of A is stored.
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the symmetric matrix A.  If UPLO = 'U', the leading
*          N-by-N upper triangular part of A contains the upper
*          triangular part of the matrix A, and the strictly lower
*          triangular part of A is not referenced.  If UPLO = 'L', the
*          leading N-by-N lower triangular part of A contains the lower
*          triangular part of the matrix A, and the strictly upper
*          triangular part of A is not referenced.
*          On exit, if INFO = 0, the factor U or L from the Cholesky
*          factorization A = U**T*U or A = L*L**T.
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value
 *                positive definite, and the factorization could not be
*                completed.

# Generate Julia method potrf!
   for  (potrf,  elty) in  # Run through 4 element types
```

6.6.3 Easy and flexible parallelism

Parallel computing remains an important research topic in numerical computing. Parallel computing has yet to reach the level of richness and interactivity required for innovation that has been achieved with sequential tools. The issues discussed in Section 6.3.1 on the balance between the human and the computer become more pronounced in the parallel setting. Part of the problem is that parallel computing means different things to different people:

1. At the most basic level, one wants instruction level parallelism within a CPU, and expects the compiler to discover such parallelism in the code. In Julia, this can be achieved explicitly with the use of the `@simd` primitive. Beyond that,

2. In order to utilize multicore and manycore CPUs on the same node, one wants some kind of multi-threading. Currently, we have experimental multi-threading support in Julia, and this will be the topic of a further paper. Julia currently does provide a `SharedArray` data structure where the same array in memory can be operated on by multiple different Julia processes on the same node.

3. Then, there is distributed memory, often considered the most difficult kind of parallelism. This can mean running Julia on anything between half a dozen to thousands of nodes, each with multicore CPUs.

In the fullness of time, there may be a unified programming model that addresses this hierarchical nature of parallelism at different levels, across different memory hierarchies.

Our experience with Star-P [**?**] taught us a valuable lesson. Star-P parallelism [**?**, **?**] (see Figure **??**) included global dense, sparse, and cell arrays that were distributed on parallel shared or distributed memory computers. Before the evolution of the cloud as we know it today, the user used a familiar front end (usually MATLAB) as the client on a laptop or desktop, and connected seamlessly to a server (usually a large distributed computer). Blockbuster functions from sparse and dense linear algebra, parallel FFTs,

parallel sorting, and many others were easily available and composable for the user. In these cases Star-P called Fortran/MPI or C/MPI. Star-P also allowed a kind of parallel for loop that worked on rows, planes or hyperplanes of an array. In these cases Star-P used copies of the client language on the backend, usually MATLAB, octave, python, or R.

Our experience taught us that while we were able to get a useful parallel computing system this way, bolting parallelism onto an existing language that was not designed for performance or parallelism is difficult at best, and impossible at worst. One of our (not so secret) motivations to build Julia was to have the right language for parallel computing.

Julia provides many facilities for parallelism, which are described in detail in the Julia manual[16]. Distributed memory programming in Julia is built on two primtives - *remote calls* that execute a function on a remote processor and *remote references* that are returned by the remote processor to the caller. These primitives are implemented completely within Julia. On top of these, Julia provides a distributed array data structure, a `pmap` implementation, and a way to parallelize independent iterations of a loop with the `@parallel` macro - all of which can parallelize code in distributed memory. These ideas are exploratory in nature, and will certainly evolve. We only discuss them here to emphasize that well-designed programming language abstractions and primitives allow one to express and implement parallelism completely within the language, and explore a number of different parallel programming models with ease. We hope to have a detailed discussion on Juila's approach to parallelism in a future paper.

We proceed with one example that demonstrates `@parallel` at work, and how one can impulsively grab a large number of processors and explore their problem space quickly.

Suppose we wish to perform a complicated histogram in parallel. We use an example from Random Matrix Theory, (but it could easily have been from finance), the computation of the scaled largest eigenvalue in magnitude of the so called stochastic Airy

[16] `http://docs.julialang.org/en/latest/manual/parallel-computing/`

operator

$$\frac{d^2}{dx^2} - x + \frac{1}{2\sqrt{\beta}} dW.$$

This is just the usual finite difference discretization of $\frac{d^2}{dx^2} - x$ with a "noisy" diagonal.

We illustrate an example of the famous Tracy-Widom law being simulated with Monte Carlo experiments for different values of the inverse temperature parameter β. The code on 1 processor is fuzzy and unfocused, as compared to the same simulation on 75

processors, which is sharp and focused.

In[73]:
```
 @everywhere begin                              # define on
every processor
 function stochastic(β=2,n=200)
   h=n^-(1/3)
   x=0:h:10
   N=length(x)
   d=(-2/h^2 .-x) +  2 sqrt(h*β)*randn(N) # diagonal
   e=ones(N-1)/h^2                            # subdiagonal
   eigvals(SymTridiagonal(d,e))[N]     # smallest negative
eigenvalue
end
end
```

In[74]:
```
t = 10000

for β=[1,2,4,10,20]
  hist([stochastic(β) for i=1:t], -4:.01:1)[2]

  plot(midpoints(-4:.01:1),z/sum(z)/.01)
end
```

ScreenShot1.png

In[75]:
```
# Readily adding 75 processors sharpens the Monte Carlo
simulation
addprocs(75)
```

In[76]:
```
t = 10000

for β=[1,2,4,10,20]
z = @parallel (+) for i=1:nprocs()
     hist([stochastic(β) for i=1:t], -4:.01:1)[2]
end

plot(midpoints(-4:.01:1),z/sum(z)/.01)
end
```

ScreenShot2.png

6.6.4 Performance Recap

In the early days of high level numerical computing languages, the thinking was that the performance of the high level language did not matter so much so long as most of the time was spent inside the numerical libraries. These libraries consisted of blockbuster algorithms that would be highly tuned, making efficient use of computer memory, cache,

and low level instructions.

What the world learned was that only a few codes were spending a majority of their time in the blockbusters. Real codes were being caught by interpreter overheads, stemming from processing more aspects of a program at run time than are strictly necessary.

As we explored in Section 6.3, one of the hindrances of completing this analysis is type information. Programming language design thus becomes an exercise in balancing incentives to the programmer to provide type information and the ability of the computer to infer type information. Vectorization is one such incentive system. Existing numerical computing languages would have us believe that this is the only system, or even if there were others, that somehow this was the best system.

Vectorization at the software level can be elegant for some problems. There are many matrix computation problems that look beautiful vectorized. These programs should be vectorized. Other programs require heroics and skill to vectorize sometimes producing unreadable code all in the name of performance. These are the ones that we object to vectorizing. Still other programs can not be vectorized very well even with heroics. The Julia message is to vectorize when it is natural, producing nice code. Do not vectorize in the name of speed.

Some users believe that vectorizing software is required to make use of special hardware capabilities including the ability to use SIMD instructions, multithreading, GPU units, and other forms of parallelism. The Julia message remains: vectorize when natural, when you feel it is right.

6.7 Conclusion and Acknowledgments

Julia was created to meet the needs of numerical computing. At the time of writing, not a day goes by where we don't learn that someone else has picked up Julia at universities and companies around the world, in fields as diverse as engineering, mathematics, physical and social sciences, finance, biotech, and many others. More than just a language, Julia

has become a place for programmers, physical scientists, social scientists, computational scientists, mathematicians, and others to pool their collective knowledge in the form of online discussions and in the form of code. Numerical computing is maturing and it is exciting to watch!

We thank Michael La Croix for his beautiful Julia display macros. The authors gratefully acknowledge support from the MIT Deshpande center for numerical innovation, the Intel Technology Science Center for Big Data, the DARPA Xdata program, the Singapore MIT Alliance, NSF Awards CCF-0832997 and DMS-1016125, VMWare Research, a DOE grant with Dr. Andrew Gelman of Columbia University for petascale hierarchical modeling, and a Citibank grant for High Performance Banking Data Analysis.

Chapter 7

Conclusion

Here is a snapshot of Julia in April, 2014

Show all the stuff that will be fun to look at because it'll be so antiquated by 2019

by following this kind of approach computer scientists can have it both ways, and make new interesting things that are also immediately useful.

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
  depth::Int   # invariant nesting depth of a Var's UnionAll
  right::Bool  # this Var is on the right-hand side of A <: B
end

type UnionSearchState
  i::Int       # (0 <= i < nbits(idxs))
  idxs::Int64  # bit vector representing combination being tested
  UnionSearchState() = new(0,0)
end

type UnionState
  depth::Int        # number of union decision points we're inside
  nnew::Int         # # unions found at next nesting depth
  stack::Vector{UnionSearchState}  # stack of decisions for each depth
  UnionState() = new(1,0,UnionSearchState[])
end

type Env
  vars::Dict{Var,Bounds}
  depth::Int
  Lunions::UnionState
  Runions::UnionState
  Env() = new(Dict{Var,Bounds}(), 1, UnionState(), UnionState())
end

issub(x, y) = forall_exists_issub(x, y, Env(), 0)
issub(x, y, env)         = (x === y)
```

```
issub(x::Ty, y::Ty, env) = (x === y) || x === BottomT

function forall_exists_issub(x, y, env, nL)
  for forall in 1:(1<<nL)
    if !isempty(env.Lunions.stack)
      env.Lunions.stack[end].idxs = forall
    end

    !exists_issub(x, y, env, 0) && return false

    if env.Lunions.nnew > 0
      push!(env.Lunions.stack, UnionSearchState())
      sub = forall_exists_issub(x, y, env, env.Lunions.nnew)
      pop!(env.Lunions.stack)
      !sub && return false
    end end
  return true
end

function exists_issub(x, y, env, nR)
  for exists in 1:(1<<nR)
    if !isempty(env.Runions.stack)
      env.Runions.stack[end].idxs = exists
    end
    for ru in env.Runions.stack; ru.i = -1; end
    for lu in env.Lunions.stack; lu.i = -1; end
    env.Lunions.depth = env.Runions.depth = 1
    env.Lunions.nnew = env.Runions.nnew = 0

    sub = issub(x, y, env)

    if env.Lunions.nnew > 0
      return true # return up to forall_exists_issub
    end
    if env.Runions.nnew > 0
      push!(env.Runions.stack, UnionSearchState())
      found = exists_issub(x, y, env, env.Runions.nnew)
      pop!(env.Runions.stack)
      if env.Lunions.nnew > 0
        return true # return up to forall_exists_issub
      end
    else
      found = sub
```

```
    end
    found && return true
  end
  return false
end

function issub_union(t::Ty, u::UnionT, env, R, state::UnionState)
  if state.depth > length(state.stack)
    # at a new nesting depth, begin by just counting unions
    state.nnew += 1
    return true
  end
  ui = state.stack[state.depth]; ui.i += 1
  state.depth += 1
  choice = u.((ui.idxs&(1<<ui.i)!=0) + 1)
  ans = R ? issub(t, choice, env) : issub(choice, t, env)
  state.depth -= 1
  return ans
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) =
  a === BottomT || 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
  a.name !== b.name && return issub(super(a), b, env)
  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
    env.depth += 1  # crossing invariant constructor, increment depth
    yes = true
    for i = 1:length(a.params)
      ai, bi = a.params[i], b.params[i]
      yes &= (issub(ai, bi, env) && issub(bi, ai, env))
    end
    env.depth -= 1
    return yes
  end
end

function issub(a::Var, b::Ty, env)
  aa = env.vars[a]
  # Vars are fully checked by the "forward" direction of A<:B in
  # invariant position.
  aa.right && return true
  return issub(aa.ub, b, env) && (env.depth == aa.depth ||
                                  issub(b, aa.lb, env))
end

lb(x, env) = isa(x,Var) ? lb(env.vars[x].lb, env) : x
ub(x, env) = isa(x,Var) ? ub(env.vars[x].ub, env) : x
join(a,b,env) = issub(a,b,env) ? b : issub(b,a,env) ? a : UnionT(a,b)

function issub(a::Union(Ty,Var), b::Var, env)
  bb = env.vars[b]
  !bb.right && return true
  if isa(a,Var)
    aa = env.vars[a]
    aa.right && return true
    # Vars must occur at same depth
    aa.depth != bb.depth && return false
    a_lb, a_ub = ub(aa.lb,env), lb(aa.ub,env)
  else
    a_lb, a_ub = a, a
```

```
  end
  b_lb, b_ub = ub(bb.lb,env), lb(bb.ub,env)
  # make sure constraint is within the current bounds of Var
  if !isa(bb.lb,Var) && !isa(bb.ub,Var)
    !issub(a_ub, b_ub, env) && return false
    if env.depth > bb.depth
      !issub(b_lb, a_lb, env) && return false
    end end

  # check & update bounds for covariant position
  # for each type a<:b, grow b's lower bound to include a, or set b's
  # lower bound equal to a typevar if its lower bound is big enough.
  if isa(a,Var) && (a === bb.lb || issub(b_lb, a_lb, env))
    bb.lb = a
  else
    !issub(a_ub, b_ub, env) && return false
    bb.lb = join(b_lb, a_ub, env)
  end

  if env.depth > bb.depth
    # check & update bounds for invariant position.
    # this would be the code for contravariant position, but since
    # we only have invariant, the covariant code above always runs too.
    if isa(a,Var) && (a === bb.ub || issub(a_ub, b_ub, env))
      bb.ub = a
    else
      # for true contravariance we would need to compute a meet here,
      # but because of invariance b_ub⊓a_lb = a_lb here always
      bb.ub = a_lb
    end end
  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, env.depth, 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) =
  a === BottomT || issub_unionall(a, b, env, true)
issub(a::UnionAllT, b::Ty, env) = issub_unionall(b, a, env, false)
```

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] Kaplan, M. A., and Ullman, J. D. A scheme for the automatic inference of variable types. *J. ACM 27* (January 1980), 128–145.

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

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

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

[15] Mohnen, M. A graphfree approach to dataflow analysis. In *Compiler Construction*, R. Horspool, Ed., vol. 2304 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 185–213.

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

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

[18] 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.

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