Abstraction in Technical Computing

by

Jeffrey Werner Bezanson

A.B., Harvard University (2004)
S.M., Massachusetts Institute of Technology (2012)

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

June 2015

Author .....
Department of Electrical Engineering and Computer Science
May 20, 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 May 20, 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.

This thesis argues 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. This thesis suggests 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 to specialize on. We show that sufficiently powerful type-based dispatch addresses these problems. The resulting language, Julia, achieves a Quine-style "explication by elimination" of many of the productive features technical computing users expect.

Thesis Supervisor: Alan Edelman
Title: Professor

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scientific computing has evolved from being essentially the only kind of computing that existed, to being a small part of how and why computers are used. Over this period of time, programming tools have advanced in terms of the abstractions and generalizations they are capable of.

Science as a whole evolves through the incorporation of specialized bits of knowledge into more powerful general theories. There is a parallel in the world of programming languages: special-purpose languages have been subsumed by more powerful and general languages. How has this trend affected scientific computing? The surprising answer is: not as much as we would like. We see scientists and technical users generally either turning away from the best new programming languages, or else pushing them to their limits in unusual ways.

In fact, we have discovered at least *49* programming languages designed for technical computing since Fortran (figure 1-1). This high number begs for an explanation. We propose that technical users crave the flexibility to pick notation for their problems, and language design — the highest level of abstraction — is where you go when you need this level of flexibility. The desire for scientists to rework their code is not so surprising when one reflects on the tradition of inventing new notation in mathematics and science.

The need for language-level flexibility is corroborated by the ways that the technical computing domain uses general purpose languages. Effective scientific libraries extensively

Productivity				Performance
Matlab	Maple	Mathematica	SciPy	Fortress
SciLab	IDL	R	Octave	Chapel
S-PLUS	SAS	J	APL	X10
Maxima	Mathcad	Axiom	Sage	SAC
Lush	Ch	LabView	O-Matrix	ZPL
PV-WAVE	Igor Pro	OriginLab	FreeMat	
Yorick	GAUSS	MuPad	Genius	
SciRuby	Ox	Stata	JLab	
Magma	Euler	Rlab	Speakeasy	
GDL	Nickle	gretl	ana	
Torch7	A+	PDL	Nial	

Figure 1-1: 49 technical computing languages classified as primarily designed for productivity or performance

employ polymorphism, custom operators, and compile time abstraction. Code generation approaches (writing programs that write programs) are unusually common. Most of these techniques are presently fairly difficult to use, and so programmers working at this level give up the notational conveniences of the purpose-built languages above.

The goal of this thesis is to identify and address the fundamental issues that have made productive and performant technical computing so difficult to achieve. We ask: which general purpose language would *also* be able to handle technical computing, such that the above workarounds would not have been necessary?

In brief, our answer is to *integrate code selection and specialization*. The aforementioned flexibility requirement can be explained as a sufficiently powerful means for *selecting* which piece of code to run. When such mechanisms are used in technical computing, there is nearly always a corresponding need to *specialize* code for specific cases to obtain performance. Polymorphic languages sometimes support forms of specialization, but often only through a designated mechanism (e.g. templates), or deployed conservatively out of resource concerns. We intend to show that an equilibrium point in the design space can be found by combining selection and specialization into a single appropriately designed dynamic type-based dispatch mechanism.

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

Table 1.1: Priorities of mainstream object-oriented and functional programming language research and implementation compared to those of the technical computing domain.

Thesis statement

Integrating code selection and specialization using type-based dynamic dispatch captures both the performance and productivity requirements of technical computing.

1.1 The technical computing problem

Table 1.1 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. 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. Of course, the features on the left side can also be useful for technical computing; we exaggerate to help make the point.

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. Imagine you want just the features on the left. Then there are many good languages available: Haskell, ML, Java, C#, perhaps C++. Getting everything on the right, by comparison, seems to be awkward. The most popular approach is to use multiple languages, as in e.g.

NumPy [112], with a high-level productivity language layered on top of libraries written in lower-level languages. Seeking a similar trade-off, others have gone as far as writing a C++ interpreter [113].

1.2 Solution space

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 [48] 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 [4] or Polyduce [25], for instance), but here we will use a dynamically typed approach.We are mostly interested in types in the sense of *data types* and *partial information* (for an excellent discussion of the many meanings of the word "type" see [64]). Additionally, some idioms in this domain appear to be inherently, or at least naturally, dynamically typed (as we will explore in chapter 2). The mechanism we will explore provides useful run time behavior and, we hope, performance improvements. Many approaches to statically checking its properties are likely possible, however we leave them to future work.

1.2.1 The importance of code selection

Most languages provide some means for selecting one of many pieces of code to run from a single point in the program. This process is often called *dispatch*. High-level languages, especially "dynamic" languages tend to provide a lot of flexibility in their dispatch systems.

This makes many sophisticated programs easier to express, but at a considerable performance cost.

There has been a large amount of work on running dynamic languages faster (notable examples are Self [31], and more recently PyPy [16]). These efforts have been very successful, but arguably leave something out: a better interface to performance. Performance-critical aspects of most dynamically typed languages (e.g. arithmetic) have fixed behavior, and it is not clear what to do when your code does not perform as well as it could. Adding type annotations can work, but raises questions about how powerful the type system should be, and how it relates to the rest of the language. We speculate that this relative lack of control keeps experts away, perpetuating the split between performance and productivity languages.

According to our theory, the issue is not really just a lack of types, but an exclusive focus on specialization, and not enough on selection. In technical computing, it is often so important to select the best optimized kernel for a problem that it is worth doing a slow dynamic dispatch in order to do so. Most of the productivity languages mentioned above are designed entirely around this principle. But their view of code selection is incomplete: the dispatch mechanisms used are ad hoc and not employed consistently. Even when an optimizing compiler for the source language is developed, it does not necessarily take all relevant forms of dispatch into account. For example, NumPy has its own dispatch system underneath Python's, and so is not helped as much as it could be by PyPy, which only understands Python method selection.

A dispatch specification, for example a method signature in a multimethod system, describes when a piece of code is applicable. A more expressive dispatch system then allows code to be written in smaller pieces, helping the compiler remove irrelevant code from consideration. It is also more extensible, providing more opportunities to identify special cases that deserve tailored implementations. Method-at-a-time JIT compilers (e.g. [54, 108]) can specialize method bodies on all arguments, and so are arguably already equipped to implement and optimize multiple dispatch (even though most object-oriented languages do not offer it). At the same time, dispatch specifications are a good way to provide type information to the

compiler.

The remaining question is what exactly we should dispatch on. Naturally, it might be nice to be able to dispatch on any criteria at all. However there is a balancing act to perform: if dispatch signatures become too specific, it becomes difficult to predict whether they will be resolved statically. At one end of the spectrum are mechanisms like static overloading in C++, where the compiler simply insists that it be possible to resolve definitions at compile time. At the other end of the spectrum, we might dispatch on arbitrary predicates (as in the SCMUTILS system [109]). We do neither of these. Instead, our dispatch is based on *sets of values that are robust under evaluation*: abstract evaluation over such sets is less likely to diverge. For example, consider the set $\{1, 2\}$. This set is not closed under "natural" functions. As we evaluate a recursive function over this abstract value, the set will typically grow, and the fixed point will be much larger than the original set. So while specializing a method for $\{1, 2\}$ could in some rare cases be useful, it also leads to unpredictable performance. To find reasonably robust sets to dispatch on, we begin with structured type tags and take the closure under data flow operations. This will be developed in detail in section 4.2.

1.2.2 Staged programming

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?

These questions lead to practical problems like ad hoc custom syntax, extra build steps, and excessive development effort (writing parsers and code generators might take more work than the core problem). If code needs to be generated based on run time information,

then a staged library might need to implement its own dispatch mechanism for invoking its generated code.

We find that our particular style of dynamic type-based dispatch provides an unusually good substrate for staged programming. The reason is that the language, though dynamically typed, has a powerful standard notion of partial information that can be used to generate code based on semantic (not just syntactic) information. In general, this semantic information can also include run time (and always, at least, approximate run time) information. Since the types used by dispatch support nested structure, they convey enough information to drive code generation for a significant class of problems. In this framework code generators are invoked automatically by the compiler, and the results can be called directly or even inlined into user code. Libraries can switch to using a staged implementation without affecting client code.

1.3 Summary of contributions

- We introduce the idea of integrated code selection and specialization. This leads to a dispatch system that provides a novel combination of performance and expressiveness.

- A type system and subtyping algorithm designed for describing method applicability and performance-critical aspects of data structures. This is the first system to apply the semantic subtyping approach to technical computing. We provide a simple implementation of our subtyping algorithm. This subtyping system is adoptable by dynamic languages that might benefit from a richer abstract value domain.

- James Morris eloquently observed that "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." [89] An additional contribution of this thesis is the application of our approach to features of technical computing environments that have not been subject to such analysis before. We provide an example application that shows how to use our language to more easily solve a difficult

computational problem.

- Finally, we contribute the Julia software system, currently a growing free software project. To our knowledge, this is the first technical computing system written in a single language. This demonstrates that our approach is both practical and appealing to a significant number of users.

Chapter 2

The nature of technical computing

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

The landscape of computing has changed dramatically over the years. Modern scientific computing environments such as MATLAB [82], R [60], Mathematica [79], Octave [90], Python (with NumPy) [112], and SciLab [53] have grown in popularity and fall under the general category known as *dynamic languages* or *dynamically typed languages*. In these 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.

2.1 What is a technical computing environment?

This question has not really been answered. In fact technical computing software has been designed haphazardly. Each system has evolved as a pile of features taken without what we consider a sufficiently critical argument.

Some languages provide a "convenient" experience that is qualitatively different from "inconvenient" languages. We believe this can be made somewhat precise. A large part of "convenience" is the reduction of the amount you need to know about any given piece of functionality in order to use it. This leads languages to adopt various forms of loose coupling, automation, and elision of software engineering distinctions that are considered important in other languages.

These systems are function-oriented, typically providing a rather large number of functions and a much smaller number of data types. Type definitions and features for organizing large systems are de-emphasized. But functions are heavily emphasized, and the functions in these systems have a particular character one might describe as "manifest": you just call them, interactively, and see what they do. This notion includes the following features:

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

Language design choices affect the ability to provide this user experience (though the first two items are also related to library design). Informally, in order to provide the desired experience a language needs to be able to assign a meaning to a brief and isolated piece of code such as `sin(x)`. This leads directly to making declarations and annotations optional, eliding administrative tasks like memory management, and leaving information implicit (for example the definition scopes and types of the identifiers `sin` and `x`). These characteristics are strongly associated with the Lisp tradition of dynamically typed, garbage collected languages with interactive REPLs.

However, there are subtle cultural differences. A case in point is the MATLAB `mldivide`, or backslash, operator [81]. By writing only `A\B`, the user can solve square, over- or under-determined linear systems that are dense or sparse, for multiple data types. The arguments

can even be scalars, in which case simple division is performed. In short, a large amount of linear algebra software is accessed via a single character! This contrasts with the software engineering tradition, where clarifying programmer intent would likely be considered more important. Even the Lisp tradition, which originated most of the convenience features enumerated above, has sought to separate functionality into small pieces. For example Scheme provides separate functions `list-ref` and `vector-ref` [3] for indexing lists and vectors.

2.1.1 Case study: Vandermonde matrices

To get a sense of how current technical computing environments work, it is helpful to look at a full implementation example. NumPy [112] provides a function for generating Vandermonde matrices which, despite its simplicity, demonstrates many of the essential language characteristics this thesis addresses. The user-level `vander` function is implemented in Python (here lightly edited for presentation):

```
def vander(x, N):
  x = asarray(x)
  if x.ndim != 1:
    raise ValueError("x must be a one-dimensional array or sequence.")

  v = empty((len(x), N), dtype=promote_types(x.dtype, int))

  if N > 0:
    v[:, 0] = 1
  if N > 1:
    v[:, 1:] = x[:, None]
    multiply.accumulate(v[:, 1:], out=v[:, 1:], axis=1)

  return v
```

This code has many interesting features. Its overall structure consists of normalizing and checking arguments, allocating output, and then calling the lower-level kernel `multiply.accumulate` (written in C) to run the inner loop. Even though most of the work is done in C, notice why this part of the code is written in Python. `vander` accepts nearly anything as its first

argument, relying on `asarray` to convert it to an array. There is no need to write down requirements on argument `x`, allowing the set of supported inputs to grow easily over time. Next, an output array of the correct size is allocated. The key feature of this line of code is that the data type to use is *computed* using the function `promote_types`. This kind of behavior is difficult to achieve in statically typed languages.

The call to `multiply.accumulate` invokes a C function called `PyUFunc_Accumulate`, which is over 300 lines long. The job of this function is to look up an appropriate computational kernel to use given the types of the input arguments and the operation being performed (`multiply`). In other words, it performs dynamic dispatch. This is notable because Python itself is already based on dynamic dispatch. However, Python's class-based dispatch system is not particularly helpful in this case, since the code needs to dispatch on multiple values. Therefore NumPy's C code implements a custom dispatch table. The targets of this dispatch are many compact, specialized loops generated by running C code through a custom preprocessor.

The mechanism is complicated, but the results are appealing: NumPy code has full run time flexibility, yet can still achieve good inner-loop performance. Notice that this result is obtained through a clever, if painstaking, mixture of binding times. Python performs some late-bound operations, then calls an early-bound C routine, which performs its own late-bound dispatch to a loop of early-bound operations.

A central message of this thesis is that this kind of program behavior is useful, highly characteristic of technical computing, and can be provided automatically at the language level. Here is how the above dispatch scheme might be implemented in Julia (mirroring the structure of the Python code as much as possible to facilitate comparison):

```
function vander(x, N::Int)
  x = convert(AbstractVector, x)
  M = length(x)
  v = Array(promote_type(eltype(x),Int), M, N)
  if N > 0
    v[:, 1] = 1
  end
  if N > 1
```

```
    for i = 2:N; v[:, i] = x; end
    accumulate(multiply, v, v)
  end
  return v
end

function accumulate(op, input, output)
  M, N = size(input)
  for i = 2:N
    for j = 1:M
      output[j,i] = op(input[j,i], input[j,i-1])
    end
  end
end
```

This code implements the entire `vander` computation (with some details left out, for example it does not provide the `axis` argument of `multiply.accumulate`). The C component is replaced by a general type-based dispatch system that handles specializing and then invoking an efficient implementation of `accumulate` for each combination of argument types.

Performance is shown in table 2.1. When repeating the computation many times for small matrix sizes, the overhead of the two-language dispatch system in NumPy completely swamps computation time. In the Julia `vander`, the type of `v` is known to the compiler, so it is able to generate a direct call to the correct specialization of `accumulate`, and there is no dispatch overhead. For large data sizes dispatch overhead is amortized, and NumPy is much faster. It is not clear why Julia is still faster in this case; it may be due to NumPy's more general algorithm that can accumulate along any dimension. In some cases, built-in routines in NumPy are slightly faster than Julia implementations. However, the Julia code for `accumulate` is so compact that it is reasonable to write manually when needed; a fully general library function is not required.

In some cases our compiler would not be able to determine the type of `v`. We simulated this situation and repeated the benchmark. Results are shown in the rightmost column of the table. Dispatch overhead is now significant for small data sizes. However, performance is still fairly good, since there is only a single layer of dispatch. The dynamic dispatch logic exists in only one place in our code, where it can be extensively optimized.

Size	# iter	NumPy	Julia	Julia with dispatch
4×4	250000	5.43	0.072	0.204
2000×2000	1	0.420	0.037	0.037

Table 2.1: Time in seconds to compute `vander`, 4×10^6 total points arranged in many small matrices or one large matrix.

There is a perception that the performance of technical code is divided into two cases: array code and scalar code. Array code is thought to perform well, and while scalar code has not traditionally performed well, it can be fixed with JIT compilation add-ons or by rewriting in C or by using extensions like Cython [116, 12]. The results here, though simple, demonstrate why we feel a different approach is needed instead: abstraction overhead is important and involves the whole program, not just inner loops.

2.2 Why dynamic typing?

Mathematical abstractions often frustrate our attempts to represent them on a computer. Mathematicians can move instinctively between isomorphic objects such as scalars and 1-by-1 matrices, but most programming languages would prefer to represent scalars and matrices quite differently. Specifying interfaces precisely seems more rigorous, and therefore possibly of use in mathematics, but technical users have voted with their code not to use languages that do this. A perennial debate occurs, for example, around the exact relationship between the notions of *array*, *matrix*, and *tensor*. In MATLAB, scalars and 1×1 matrices are indistinguishable. Informal interfaces, e.g. as provided by "duck typing", are a viable way to share code among disciplines with different interpretations of key abstractions.

2.2.1 Mismatches with mathematical abstractions

Programs in general, deal with values of widely varying disjoint types: functions, numbers, lists, network sockets, etc. Type systems are good at sorting out values of these different types. However, in mathematical code most values are numbers or number-like. Numerical properties (such as positive, negative, even, odd, greater than one, etc.) are what matter, and these are highly dynamic.

The classic example is square root (`sqrt`), whose result is complex for negative arguments. Including a number's sign in its type is a possibility, but this quickly gets out of hand. When computing eigenvalues, for example, the key property is matrix symmetry. Linear algebra provides many more examples where algorithm and data type changes arise from subtle properties. These will be discussed in more detail in section 5.4.

We must immediately acknowledge that static type systems provide mechanisms for dealing with "dynamic" variations like these. However, these mechanisms require deciding which distinctions will be made at the type level, and once the choice is made we are restricted in how objects of different types can be used. In the domain of linear algebra, for example, it would be useful to have matrices that are considered different types for some purposes (e.g. having different representations), but not others (e.g. different types of matrices can be returned from the same function).

2.2.2 I/O and data-oriented programming

Inevitably there is a need to refer to a datatype at run time. A good example is file I/O, where you might want to say "read n double precision numbers from this file". In static languages the syntax and identifiers used to specify data types for these purposes are usually different from those used to specify ordinary types. A typical example is a constant like `MPI_DOUBLE` [105], used to indicate the data type of an MPI message. Many libraries, especially those for tasks like interacting with databases and file formats, have their own set of such tags. Using more than one of these libraries together can result in boilerplate code to map between tag systems.

Polymorphic languages provide better interfaces in these cases. For example, C++ overloading can be used to provide the correct tag automatically based on the type of a function argument. However, these techniques only work for statically known types. Code like that in the `vander` example that determines types using arbitrary computation is excluded. Dynamic typing provides a complete solution. Indeed, I/O examples are often used to motivate dynamic typing [2]. With dynamic typing, a single set of names can be used to refer to all types regardless of binding time, and the user may write whatever code is convenient to handle classifying data moving in and out of a program.

2.3 Trade-offs in current designs

Current systems accept certain design trade-offs to obtain flexibility.

2.3.1 Vectorization

The attitude of vectorization proponents has been somewhat famously summarized as "Life is too short to spend writing for loops." [1] The vectorized style has the following touted benefits:

1. One may write `sin(x)` to compute the sine of all elements of `x`, which is more compact than writing a loop.

2. Most execution time can be spent inside an expertly written library, taking advantage of special hardware (e.g. SIMD units or stream processors like GPUs) and parallelism implicitly.

3. The performance of the high-level language being used becomes less relevant.

4. The compiler can optimize across whole-array operations, potentially drastically rearranging computations in a way that would be difficult to do by examining one scalar operation at a time.

Under close inspection, this argument begins to break down.

An equivocation takes place between being *able* to write a loop as `sin(x)` and being *required* to. The former is all we really want, and it is possible in any language that supports some form of function overloading. There is no inherent reason that a high-performance language like C++ or Fortran cannot also support more compact, high-level notation.

Studies reveal that many real-world applications written in vectorization languages actually do not spend most of their time in libraries [88]. It is possible for an interpreter to be so slow that it cancels the advantage of optimized kernels for a range of realistic data sizes.

The final reason, enabling advanced optimizations, is compelling. However at a certain point a "vectorization bottleneck" can be reached, where performance is limited by expressing computations in terms of in-memory arrays. Figure 2-1 compares the operation rate of three different algorithms for evaluating the polynomial $x^2 + x - 1$ over double precision vectors of varying size. The solid line describes a naive algorithm, where each operation allocates a new array and fills it with one intermediate result (a total of three arrays are needed for the full evaluation). The dotted line algorithm is similar in spirit, except the three operations are "fused" into a single loop. Only one pass is made over the data and one new array is allocated. Naturally performance is much better in this case, and advanced array programming environments like APL [43] and ZPL [71] implement this optimization.

However, this level of performance is not impressive compared to the dashed line, which describes the operation rate for computing the same results but summing them instead of storing them. Finally, the dash-dotted line uses no memory at all, instead computing input data points from a linear range. Of course a real use case might require data to be read from memory or stored, so this comparison is unfair (though not entirely, since many numerical environments have represented even simple objects like linear ranges using dense vectors [80]). Still the point should be clear: vectors are not necessarily a good abstraction for high performance. The observation that memory is a bottleneck is not new, but for our purposes the important point is that languages that only support vectorized syntax cannot express the code rearrangements and alternate algorithms that can be necessary to get the

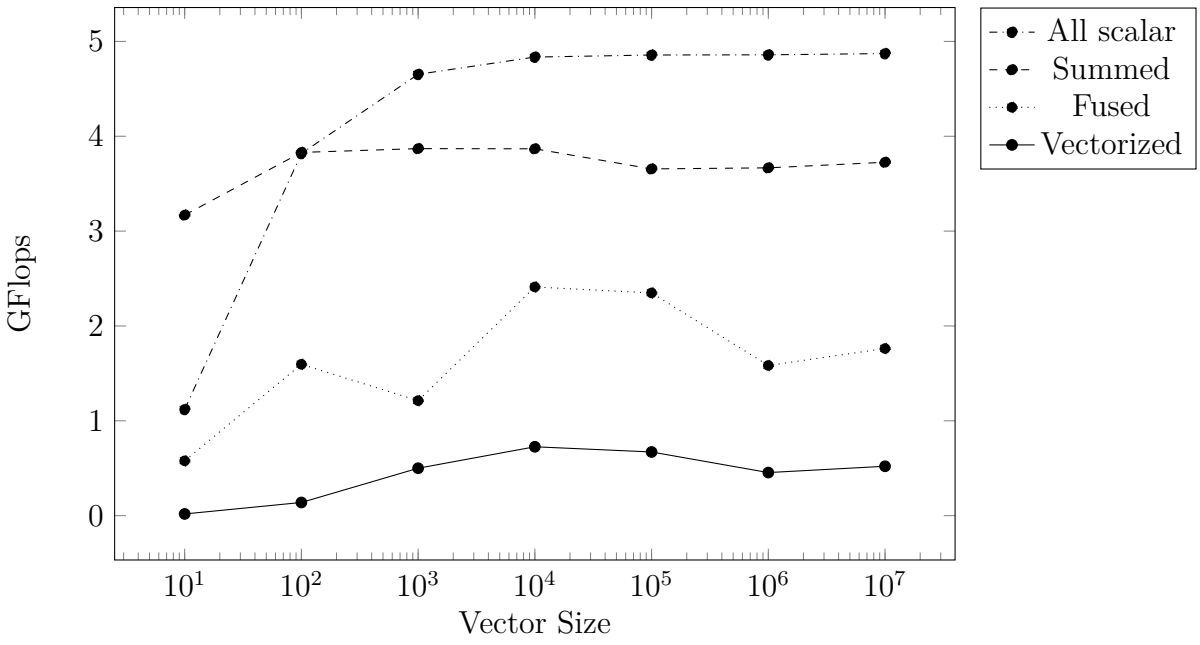

Figure 2-1: Performance of evaluating $x^2 + x - 1$ over a double precision vector of length n. Operation repeated $10^7/n$ times. (Core i7-3517U 1.9GHz, 4MB cache, 8GB RAM)

best performance. A good technical computing language needs a more general interface to performance.

The performance model resulting from vectorized syntax can be difficult to understand, since it is difficult for users to predict which optimizations will be applied. A good example was discussed on the `julia-users` mailing list [83]. Jason Merrill discovered a Mathematica [79] list discussion of the fastest way to count how many values fall within a given range. Of many proposed implementations, the fastest turned out to be

```
Total@Unitize@Clip[data,{min,max},{0,0}]
```

Arguably, it is difficult to know that this will run faster than, for example,

```
Count[data, x_/; min<=x<=max]
```

2.3.2 Data representation

There are two broad categories of programming language data models. The first, favored by static languages, provides structures that are complex and rigid but efficient. Specific data types can be grouped and stored together with known offsets and alignments. Most data is stored directly, and pointer indirection must be explicitly requested. The second kind of data model, favored by dynamic languages, prioritizes flexibility and simplicity. Most data consists of pointers to tagged objects. Indirection is the default.

It's reasonable to guess that these models would trade off about equally. If you want the flexibility to substitute any kind of value in any location at run time, making everything a pointer is the best you can do. The data layouts used by static languages require some amount of metadata to describe, and manipulating this metadata at run time could have significant overhead. In the "dynamic" data model, even though everything is a pointer, at least the code *knows* that everything is a pointer, and so does not waste time switching between different representations.

Interestingly, this intuition turns out not to be accurate. As memory performance becomes a larger bottleneck, using less space and avoiding indirection becomes more important. Furthermore, most data in dynamic language programs is not as dynamic as its authors might think, with homogeneous arrays being quite common. Good evidence for this claim was documented as part of work on *storage strategies* [17], a technique for adapting dynamic languages to use more efficient representations. The authors found that it was worthwhile to represent an array as a wrapper providing a heterogeneous array interface on top of a homogeneous array of varying type, even at the expense of extra dispatch and occasionally changing the representation.

In some sense, our work takes this idea to its logical conclusion: why not support a range of data representations from the beginning, managed by a single sufficiently powerful dispatch system?

2.4 A compiler for every problem

In several prominent cases the language of a scientific sub-discipline is so far removed from conventional programming languages, and the need for performance is so great, that practitioners decide to write their own compilers. For example, the Tensor Contraction Engine [10] accepts a mathematical input syntax describing tensor operations used for electronic structure calculations in quantum chemistry. The software performs many custom optimizations and finally generates efficient parallel Fortran code. Other examples of such scientific code generators include Firedrake [95] and FEniCS [74] for FEM problems, PyOP2 [96] for general computations on meshes, FFTW [46] for signal processing, and Pochoir [110] for stencil computations.

These systems can be highly effective, offering performance and productivity for their problem domains that are not matched elsewhere. In the long term, however, a proliferation of such projects would not be a good thing. It is substantially harder to reuse code between languages and compilers than to reuse library code within a language, and it is harder to develop a compiler than a library. Mainstream languages have not been doing enough to facilitate these use cases. It would not be realistic to expect to replace all of these systems at once, but Julia's style of type-based dispatch helps. This will be discussed further in section 4.5.

2.5 Social phenomena

In the open source world, the architecture of a system can have social implications. Maintaining code in multiple languages, plus interface glue between them, raises barriers to contribution. Making the high-level language fast helps tremendously here, but there are also cultural differences between programmers who understand machine details and want more control, and those who are happy to use defaults chosen by somebody else. We will argue that it is possible to blend these styles, creating an effective compromise between the two camps. Evan Miller expressed this well as "getting disparate groups of programmers to code

in the same language...With Julia, it's the domain experts and the speed freaks." [84]

This cultural difference often appears in the form of a decision about what to express at the type level. For example, imagine a user has linear algebra code that decides whether to use the QR factorization or the LU factorization to solve a system:

```
if condition
    M = qr(A)
else
    M = lu(A)
end
solve(M, x)
```

Initially, `qr` and `lu` might both return pairs of matrices. Later, linear algebra library developers might decide that each function should return a specialized type to store its factorization in a more efficient format, and to provide dispatch to more efficient algorithms. This decision should not break the user's intuition that the objects are interchangeable in certain contexts. We will discuss more extensive examples of such situations in chapter 5.

Chapter 3

Design space

This chapter will attempt to locate technical computing within the large design space of programming languages. It will discuss how the priorities introduced in section 1.1 relate to type systems and dispatch systems. Our lofty goal is to preserve the flexibility and ease of popular high level languages, while adding the expressiveness needed for performant generic programming.

3.1 A binding time dilemma

Consider the following fragment of a function called `factorize`, which numerically analyzes a matrix to discover structure that can be exploited to solve problems efficiently:

```
if istril(A)
    istriu(A) && return Diagonal(A)
    utri1 = true
    for j = 1:n-1, i = j+1:m
        if utri1 && A[i,j] != 0
            utri1 = i == j + 1
        end
    end
    utri1 && return Bidiagonal(diag(A), diag(A, -1), false)
    return LowerTriangular(A)
end
```

The code returns different structured matrix types based on the input data. This sort of pattern can be implemented in object-oriented languages by defining an interface or base class (perhaps called `StructuredMatrix`), and declaring `Diagonal`, `Bidiagonal`, and so on as subclasses.

Unfortunately, there are several problems with this solution. Class-based OO ties together variation and dynamic dispatch: to allow a value's representation to vary, each use of it needs to perform an indirect call. A major reason to use structured matrix types like `Bidiagonal` is performance, so we'd rather avoid the overhead of indirect calls. Most code operating on `Bidiagonal` would be implementing algorithms that exploit its representation. These algorithms would have to be implemented as methods of `Bidiagonal`. However this is not natural, since one cannot expect to have every function that might be applied to bidiagonal matrices defined in one place. This kind of computing is function-oriented.

Let's try a different abstraction mechanism: templates and overloading. That would allow us to write code like this:

```
if diagonal
    solve(Diagonal(M), x)
elseif bidiagonal
    solve(Bidiagonal(M), x)
end
```

Here, structured matrix types are used to immediately select an implementation of `solve`. But the part that selects a matrix type can't be abstracted away. We would like to be able to write `solve(factorize(M), x)`.

Given these constraints, the traditional design of technical computing systems makes sense: pre-generate a large number of optimized routines for various cases, and be able to dispatch to them at run time.

It's clear what's going on here: mathematics is dependently typed. If a compiler could prove that a certain matrix were bidiagonal, or symmetric, there would be no problem. But knowing what a given object *is* in a relevant sense can require arbitrary proofs, and in research computing these proofs might refer to objects that are not yet well understood.

Most languages offer a strict divide between early and late binding, separated by what the type system can statically prove. The less the type system models the domain of interest, the more it gets in the way. The compiler's ability to generate specialized code is also coupled to the static type system, preventing it from being reused for domains the type system does not handle well. In general, of course, lifting this restriction implies a need to generate code at run time, which is not always acceptable. However run time code generation might not be necessary in every case. A compiler could, for example, automatically generate a range of specialized routines and select among them at run time, falling back to slower generic code if necessary.

Consider the trade-off this entails. The programmer is liberated from worrying about the binding time of different language constructs. With a JIT compiler, performance will probably be good. But without JIT, performance will be unpredictable, with no clear syntactic rules for what is expected to be fast. So it is not surprising that software engineering languages shun this trade-off. However we believe it is the trade-off technical computing users want by default, and it has not generally been available in any language.

Designing for this trade-off requires a change of perspective within the compiler. Most compilers focus on, and make decisions according to, the language's source-level type system. Secondarily, they will perform flow analysis to discover properties not captured by the type system. Given our goals, we need to invert this relationship. The focus needs to be on understanding program behavior *as thoroughly as possible*, since no one set of rules will cover all cases. Usually changing a language's type system is a momentous event, but in the analysis-focused perspective the compiler can evolve rapidly, with "as thoroughly as possible" handling more cases all the time. This approach is still compatible with static typing, although we will not discuss it much. Nothing stops us from picking a well-defined subset of the system, or a separate type system, to give errors about.

The notion of "as thorough as possible" analysis is formalized by domain theory.

3.2 Domain theory

In the late 1960s Dana Scott and Christopher Strachey asked how to assign meanings to programs, which otherwise just appear to be lists of symbols [102]. For example, given a program computing the factorial function, we want a process by which we can assign the meaning "factorial" to it. This led to the invention of domain theory, which can be interpreted as modeling the behavior of a program without running it. A "domain" is essentially a partial order of sets of values that a program might manipulate. The model works 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. This model provides a way to think about the meaning of a program without running the program. Even the "result" of a non-terminating program has a representation (the bottom element).

The idea of "running a program without running it" is of great interest in compiler implementation. A compiler would like to discover as much information as possible about a program without running it, since running it might take a long time, or produce side effects that the programmer does not want yet, or, of course, not terminate. The general recipe for doing this is to design a partial order (lattice) that captures program properties of interest, and then describe all primitive operations in the language based on how they transform elements of this lattice. Then an input program can be executed, in an approximate sense, until it converges to a fixed point. Most modern compilers use a technique like this (sometimes called abstract interpretation [35]) to semi-decide interesting properties like whether a variable is a constant, whether a variable might be used before it is initialized, whether a variable's value is never used, and so on.

Given the general undecidability of questions about programs, analyses like these are *conservative*. If our goal is to warn programmers about use of uninitialized variables, we want to be "better safe than sorry" and print a warning if such a use *might* occur. Such a use corresponds to any lattice element greater than or equal to the element representing "uninitialized". Such conservative analyses are the essence of compiler transformations

for performance (optimizations): we only want to perform an optimization if the program property it relies on holds for sure.

The generality of this approach allows us to discover a large variety of program properties as long as we are willing to accept some uncertainty. Even if static guarantees are not the priority, or if a language considers itself unconcerned with "types", the domain-theoretic model is still our type system in some sense, whether we like it or not (as implied by [101]).

3.2.1 Programs and domains

Program analyses of this kind have been applied to high-level languages many times. The common pitfall is that the analysis can easily *diverge* for programs that make distinctions not modeled by the lattice.

Consider the following program that repeatedly applies elementwise exponentiation (`.^`) to an array:

```
A = [n]
for i = 1:niter
    A = f(A .^ A)
end
return A
```

We will try to analyze this program using the lattice in figure 3-1.

The result depends strongly on how the `.^` and `^` functions are defined (assume that `.^` calls `^` on every element of an array). The code might look like this:

```
function ^(x, y)
    if trunc(y) == y
        if overflows(x, y)
            x = widen(x)
        end
        # use repeated squaring
    else
        # floating point algorithm
    end
end

function f(a)
```

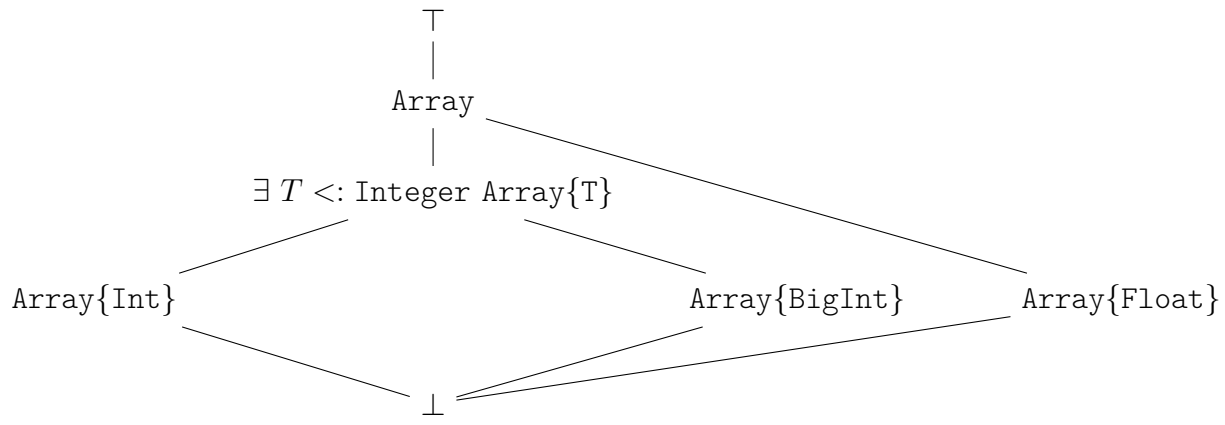

Figure 3-1: A lattice of array types

```
    if all(trunc(y) .== y)
        # integer case
    else
        # general case
    end
end
```

This code implements the user-friendly behavior of automatically switching to the `BigInt` type on overflow, by calling `widen`.

Assume we initially know that `n` is an `Int`, and therefore that `A` is an `Array{Int}`. Although `A` really does have this type (even if the type is only an artifact of the analysis), the code does not mention types anywhere. Next, the type of an element taken from `A` (`Int`) will flow to the `y` argument of `^`. The function's behavior crucially depends on the test `trunc(y) == y`. It is always true for integer arguments, but it is unlikely that the analysis can determine this. Therefore `^` might return an `Int`, `BigInt`, or `Float`, and we can only conclude that `A.^A` is an `Array`. When function `f` is called, it performs a fairly expensive test to see if every element is an integer. Only through human reasoning about the whole program do we see that this test is unnecessary. However when the analysis considers `f` applied to type `Array`, our type information is likely to diverge further to $\top$, since we don't know anything about the array's elements.

The problem here is that the program is not written in terms of the underlying value domain, even though it could be. We might have written `isa(y,Integer)` instead of `trunc(y) == y`. However, there are reasons that code like this might exist. The programmer might not be aware of the type system, or the conditional might refer to a property that was not originally reflected in the type system but is later, as the language and libraries evolve. Adding type declarations is not an ideal solution since it restricts polymorphism, and the programmer might not know the type of `A.^A` or even the type of `n`.

Our solution is to give some simple tools to the library developer (who, of course, is not necessarily a different person). In Julia, one can implement `^` and `f` as follows:

```
function ^(x, y)
    # original code
end

function ^(x, y::Integer)
    if overflows(x, y)
        x = widen(x)
    end
    # ...
end

function f(a)
    # original code
end

function f{T<:Integer}(a::Array{T})
    # integer case
end
```

Now, from the same source program, we can conclude that only `^(x, y::Integer)` is applicable, so we know that the result of `A.^A` is some integer array. The library writer can intercept this case with the definition `f{T<:Integer}(a::Array{T})`[1], avoiding the check `all(trunc(y) .== y)`. Since we started with an integer array, the whole program behaves exactly as before.

[1] Although the signature of this method happens to match one of the lattice elements in figure 3-1, this is a coincidence. Method applicability is determined dynamically.

We have neither added restrictions, nor asked for redundant type annotations. All we did is add a dispatch system, and encourage people to use it. Using dispatch in this way is optional, but comes with benefits for structuring programs. For example, the function `f` above might perform much better on integer arrays. We can implement this case separately from the checks and extra logic that might be needed for other data types, leading to cleaner code and also making the core definition of `f` smaller and therefore more likely to be inlined.

The original version of this code can also be written in our system, though it probably will not perform as well. However, it might still perform better than one would expect, since the functions that it calls are in turn defined using the same dispatch mechanism. This mechanism is expressive enough to be used down to the lowest levels of the system, providing leverage one would not get from an object system or performance add-on.

Having type information attached to user or library definitions increases the value of *reverse* data flow analysis. One way to describe why analyses of highly dynamic code diverge is that everything in the program monotonically increases the number of cases we need to consider; there is no way to narrow possibilities. But if we know that a function is only defined on type `T`, every value that passes through its argument list is narrowed to `T`.

3.3 Dispatch systems

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

However in class-based OO there is essentially *no way* to create an operation that dis-

patches on existing types. This has been called "the expression problem" [114]). While many kinds of object-oriented programs can ignore or work around this problem, technical programs cannot. In this domain most programs deal with the same few types (e.g. numbers and arrays), and might sensibly want to write new operations that dispatch on them.

3.3.1 A hierarchy of dispatch

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 [2]:

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

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

```
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)
        a.encounter(b)
    end
end
```

Here all of the simulation's animal behavior has been 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` could be 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. Here `A<:B` is used to declare a subclass relationship; it says that an `A` is a kind of `B`:

```
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 still more can be done. Notice that in the first transformation 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 begs a question: why is $n = 2$ special? It isn't, and we could 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 is sometimes called *external dispatch*. This change has 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 for *any number* of objects:

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

We can also abstract over more subtle properties, like whether the arguments are two animals of the same type:

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

3.3.2 Predicate dispatch

Predicate dispatch is a powerful object-oriented mechanism that allows methods to be selected based on arbitrary predicates [42]. 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 [106], 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.[3]

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 useful, and could be combined with a language like Julia or, indeed, most object-oriented languages

[3] Many type systems involving bounded quantification, such as system $F_{<:}$, are already undecidable [93]. However, they seem to terminate for most practical programs, and also admit minor variations that yield decidable systems [26]. It is fair to say they are "just barely" undecidable, while predicates are "very" undecidable.

(this has been done for Java [85]). However this comes at the expense of making predicates second-class citizens of the type system.

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. With predicate dispatch, the first version of the code in section 3.2.1 could have been rearranged to use dispatch instead of `if` statements. This might have advantages for readability and extensibility, but not for performance.

3.3.3 Symbolic programming

There has always been a divide between "numeric" and "symbolic" languages in the world of technical computing. To many people the distinction is fundamental, and we should happily live with both kinds of languages. But if one insists on an answer as to which approach is the right one, then the answer is: symbolic. Programs are ultimately symbolic artifacts. Computation is limited only by our ability to describe it, and symbolic tools are needed to generate, manipulate, and query these descriptions. For example in numerical computing, a successful approach has been to design high-performance kernels for important problems. From there, the limiting factor is knowing *when* to use each kernel, which can depend on many factors from problem structure to data size. Symbolic approaches can be used to automate this. We will see some examples of this in chapter 5.

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. Depending on details, their power is roughly equal to that of predicate dispatch.

However, symbolic programming lacks data abstraction: the concrete representations of values are exposed to the dispatch system. In such a system, there is no difference between being a list and being something *represented* as a list. If the representation of a value changes,

	Domain	Dynamic	Spec.	Pattern	S.T.S.	S.C.
Methods	$O(1)$				✓	✓
Virtual methods	$O(1)$	✓			✓	✓
Overloading	$O(n)$				✓	✓
Templates	$O(n)$		✓		✓	
Closures	$O(1)$	✓			✓	✓
Duck typing	$O(1)$	✓				✓
Multimethods	$O(n)$	✓			?	?
Predicate dispatch	$O(nm)$	✓		1	?	?
Typeclasses	$O(m)$	2	3		✓	✓
Term rewriting	$O(nm)$	✓		✓		
Julia	$O(nm)$	✓	✓		?	

Table 3.1: Attributes of several code selection features. Spec. stands for specialization. S.T.S. stands for statically type safe. S.C. stands for separate compilation. 1. Depending on design details, 2. When combined with existential types, 3. Optionally, with declarations,

the value can be inadvertently "captured" by a dispatch rule that was not intended to apply to it, violating abstraction.

3.3.4 Choices, choices

Table 3.1 compares 11 language features. Each provides some sort of control flow indirection, packaged into a structure designed to facilitate reasoning (ideally human reasoning, but often the compiler's reasoning is prioritized). The "domain" column describes the amount of information considered by the dispatch system, where n is the number of arguments and m is the amount of relevant information per argument. $O(1)$ means each use of the feature considers basically the same amount of information. $O(n)$ extends the process to every argument. $O(m)$ means that one value is considered, but its structure is taken into account.

Squares with question marks are either not fully understood, or too sensitive to design details to answer definitively.

This table illustrates how many combinations of dispatch semantics have been tried. Keeping track of these distinctions is a distraction when one is focused on a non-programming problem. Including more than one row of this table makes a language especially complex and difficult to learn.

3.4 Subtyping

So far we have seen some reasons why dispatch contributes significantly to flexibility and performance. However we have only actually dispatched on fairly simple properties like whether a value is an integer. How powerful should dispatch be, exactly? Each method signature describes some set of values to which it applies. Some signatures might be more specific than others. The theory governing such properties is subtyping.

It turns out that a lot of relevant work on this has been done in the context of type systems for XML [57, 13]. XML at first seems unrelated to numerical computing, and indeed it was quite a while before we discovered these papers and noticed the overlap. However if one substitutes "symbolic expression" for "XML document", the similarity becomes clearer. In XML processing, programs match documents against patterns in order to extract information of interest or validate document structure. These patterns resemble regular expressions, and so also denote sets.

In our context, some argument lists and the properties of some data structures are sufficiently complex to warrant such a treatment. For example, consider a `SubArray` type that describes a selection or "view" of part of another array. Here is part of its definition in Julia's standard library:

```
const ViewIndex = Union(Int, Colon, Range{Int}, UnitRange{Int},
                        Array{Int,1})
immutable SubArray{T, N, P<:AbstractArray,
                   I<:Tuple{ViewIndex...}} <: AbstractArray{T,N}
```

The properties of the `SubArray` type are declared within curly braces. A `SubArray` has an element type, number of dimensions, underlying storage type, and a tuple of indexes (one index per dimension). Limiting indexes to the types specified by `ViewIndex` documents what kind of indexes can be supported efficiently. Different index tuples can yield drastically different performance characteristics.

Without the ability to describe these properties at the type level, it would be difficult to implement an efficient `SubArray`. In section 3.2.1 we only needed to test fairly simple conditions, but the checks here would involve looping over indexes to determine which dimensions to drop, or to determine whether stride-1 algorithms can be used, and so on.

3.5 Specialization

3.5.1 Parametric vs. ad hoc polymorphism

The term *polymorphism* refers generally to reuse of code or data structures in different contexts. A further distinction is often made between *parametric* polymorphism and *ad hoc* polymorphism. Parametric polymorphism refers to code that is reusable for many types because its behavior does not depend on argument types (for example, the identity function). Ad hoc polymorphism refers to selecting *different* code for different circumstances.

In theory, a parametrically polymorphic function works on all data types. In practice, this can be achieved by forcing a uniform representation of data such as pointers, which can be handled by the same code regardless of what kind of data is pointed to. However this kind of representation is not the most efficient, and for good performance specialized code for different types must be generated.

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 or types. These specializations could be stored in a lookup table, for use either at compile time or at run time.

Next we can imagine that the compiler might not optimally specialize certain definitions,

and that the programmer would like to provide hints or hand-written implementations to speed up special cases. For example, imagine a function that traverses a generic array. A compiler-generated specialization might inline a specific array type'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.

3.5.2 Separate compilation

Writing the signature of a generic method that needs to be separately compiled, as in Java, can be a difficult exercise. The programmer must use the type system to write down sufficient conditions on all arguments. The following example from a generic Java graph library [50] demonstrates the level of verbosity that can be required:

```
public class breadth_first_search {
  public static<Vertex, Edge extends GraphEdge<Vertex>,
                VertexIterator extends Iterator<Vertex>,
                OutEdgeIterator extends Iterator<Edge>,
                Visitor extends BFSVisitor,
                ColorMap extends ReadWriteMap<Vertex, Integer>>
  void go(VertexListAndIncidenceGraph<Vertex,Edge,
            VertexIterator,VerticesSizeType,OutEdgeIterator,
            DegreeSizeType> g,
          Vertex s, ColorMap c, Visitor vis);
}
```

If, however, we are going to specialize the method, the compiler can analyze it using actual types from call sites, and see for itself whether the method works in each case. This is how C++ templates are type checked; they are analyzed again for each instantiation.

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

3.6 Easy polymorphism

The total effect of these design considerations is something more than the sum of the parts. We obtain what might be called "easy polymorphism", where many of the performance and expressiveness advantages of generic programming are available without their usual complexity. This design arises from the following five part recipe:

1. One mechanism for overloading, dispatch, and specialization
2. Semantic subtyping
3. Make type parameters optional
4. Code specialization by default
5. Types as ordinary values

The first three items provide three forms of "gradualism": adding methods, adding more detailed type specifications to arguments, and adding parameters to types. The last two items alleviate some of the difficulty of type-level programming. The last item is especially controversial, but need not be. Consistent and practical formulations allowing types to be values of type `Type` have been demonstrated [21].

Chapter 4

The Julia approach

4.1 Core calculus

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

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

The `new` construct creates a value from a type tag and some other values; it resembles the `Dynamic` constructor in past work on dynamic typing [2]. The tag is determined by evaluating an expression. This means constructing types is part of programming, and types

can contain nearly arbitrary data.[1] In Julia syntax, types are constructed using curly braces; this is shorthand for an appropriate `new` expression (tags are a specific subset of data values whose tag is the built-in value `Tag`). Although type expressions are quite often constants, one might also write `MyType{x+y}`, to construct a `MyType` with a parameter determined by computing `x+y`. This provides a trade-off that we feel is highly desirable for our intended applications:

- The compiler cannot always statically determine types.
- Program behavior does not depend on the compiler's (in)ability to determine types.
- The compiler can do whatever it wants in an attempt to determine types.

The last point means that if the compiler is somehow able to statically evaluate `x+y`, it is free to do so, and gain sharper type information. This is a general property of dynamic type inference systems [62, 77, 11, 8, 5]. Julia's types are designed to support this kind of inference, but they are also used for code selection and specialization regardless of the results of inference.

In practice, `new` is always wrapped in a constructor function, abstracting away the inconvenience of constructing both a type and a value. In fact, `new` for user-defined data types is syntactically available only within the code block that defines the type. This provides some measure of data abstraction, since it is not possible for user code to construct arbitrary instances.

Constants are pre-built tagged values.

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

[1] We restrict this to values that can be accurately compared by `===`, which will be introduced shortly.

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

The last item is the special abstract variadic tuple type.

Data model

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

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

where bit^n represents a string of n bits, and ref represents a reference to a tagged data value. Data is automatically laid out according to the platform's application binary interface (ABI). Currently all compound types and heterogeneous tuples use the C struct ABI, and homogeneous tuples use the array ABI. Data may be declared mutable, in which case its representation is implicitly wrapped in a mutable cell. A built-in primitive equality operator `===` is provided, based on *egal* [9] (mutable objects are compared by address, and immutable objects are compared by directly comparing both the tag and data parts bit-for-bit, and recurring through references to other immutable objects).

Functions and methods

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

and its most specific method matching the tag of argument e_2 is called.

We use the keyword `function` for method definitions for the sake of familiarity, though `method` is arguably more appropriate. Method definitions subsume lambda expressions. Each method definition modifies a generic function named by the argument x_{name}. The generic function to extend is specified by its name rather than by the value of an expression in order to make it easier to syntactically restrict where functions can be extended. This, in turn, allows the language to specify when new method definitions take effect, providing useful windows of time within which methods do not change, allowing programs to be optimized more effectively (and hopefully discouraging abusive and confusing run time method replacements).

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

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

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

4.2 Type system

Our goal is to design a type system 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 [47, 23]. Since our goals only include performance and expressiveness, we simply skip the later steps for now, and do not consider in detail possible approaches to type checking. A good slogan for this attitude might be "evaluate softly and carry a big subtype relation."

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 composability). These requirements are fairly strict. Beginning with the simplest possible system, we added features as needed to satsify the aforementioned closure properties.

We will define our types by 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. Also note that the all-important array type is written as `Array{T,N}` where `T` is an element type and `N` is a number of dimensions.

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

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

These types closely resemble the classes of an object-oriented language with generic (parametric) types, invariant type parameters, and no concrete inheritance. We prefer parametric *invariance* partly for reasons that have been addressed in the literature [36]. Invariance preserves the property that the only subtypes of a concrete type are `Bottom` and itself. This is important given how we map types to data representations: an `Array{Int}` cannot also be an `Array{Any}`, since those types imply different representations (an `Array{Any}` is an array of pointers). Tuples are a special case where covariance works, because each component type need only refer to single value, so there is no need for concrete tuple types with non-concrete parameters.

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

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

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

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 [70]. This formulation has the advantage that *any* type that is a subtype of a tuple type can be used to express the signature of a method. It also makes the system simpler and more reflective, since subtype queries can be

used to ask questions about methods.

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

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

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

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

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

This is one of several 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 many static type checkers. Yet from the perspectives of both data flow analysis and method specialization, they are perfectly natural and even essential [35, 59, 104].

The next problem we need to solve arises from data flow analysis of the `new` construct. When a type constructor `C` is applied to a type S that is known only approximately at compile time, the type $\texttt{C}\{S\}$ does not correctly represent the result. The correct result would be the union of all types $\texttt{C}\{T\}$ where $T \leq S$. 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 within specified bounds:

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

where $A[T/\texttt{T}]$ means T is substituted for `T` in expression A.

This is similar to an existential type [22]; 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. However we will still use $\exists$ notation as a shorthand.

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.

Type system variants

Our design criteria do not identify a unique type system; some variants are possible. The following features would probably be fairly straightforward to add:

- Structurally subtyped records
- μ-recursive types (regular trees)

- General regular types (allowing ... in any position)

The following features would be difficult to add, or possibly break decidability of subtyping:

- Arrow types
- Negations
- Intersections, multiple inheritance
- Universal quantifiers
- Contravariance

4.2.1 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.

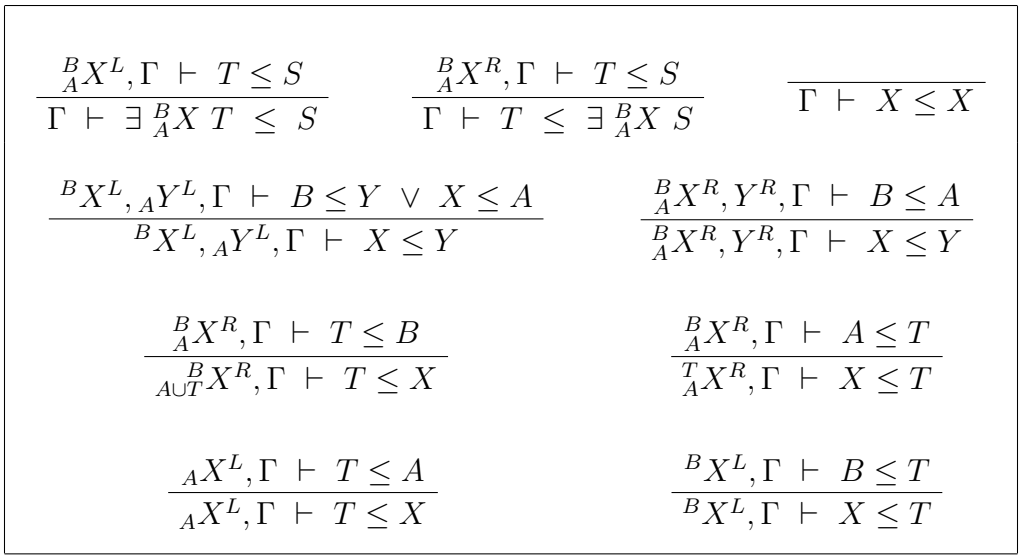

Figure 4-1: Subtyping algorithm for `UnionAll` ($\exists$) and variables. X and Y are variables, A, B, T, and S are types or variables. ${}^{B}_{A}X$ means X has lower bound A and upper bound B. ^{R} and ^{L} track whether a variable originated on the right or on the left of $\leq$. Rules are applied top to bottom.

4.2.2 Subtyping

Computing the subtype relation is the key algorithm in our system. It is used in the following cases:

- Determining whether a tuple of arguments matches a method signature.
- Comparing method signatures for specificity and equality.
- Source-level type assertions.
- Checking whether an assigned value matches the declared type of a location.
- Checking for convergence during type inference.

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

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

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

The algorithm for `UnionAll` and variables is shown in figure 4-1. The first row says to handle a `UnionAll` by extending the environment with its variable, marked according to which side of $\leq$ it came from, and then recurring into the body. In analogy to union types, we need to check that for all variable values on the left, there exists a value on the right such that the relation holds. The for-all side is relatively easy to implement, since we can just use a variable's bounds as proxies for its value (this is shown in the last row of the figure). We implement the there-exists side by narrowing a variable's bounds (raising the lower bound and lowering the upper bound, in figure row 3). The relation holds as long as the bounds remain consistent (i.e. lower bound $\leq$ upper bound). The algorithm assumes that all input types are well-formed, which includes variable lower bounds always being less than or equal to upper bounds.

The rules in the second row appear asymmetric. This is a result of exploiting the lack of contravariant constructors. No contravariance means that every time a right-side variable appears on the *left* side of a comparison, it must be because it occurs in invariant position, and the steps outlined in the first paragraph of this section have "flipped" the order (comparing both `B` $\leq$ `C` and `C` $\leq$ `B`). This explains the odd rule for comparing two right-side variables: this case can only occur with differently nested `UnionAll`s and invariant constructors, in which case the relation holds only if all involved bounds are equal. By symmetry, one would

expect the rule in row 3, column 2 to update X's upper bound to $B \cap T$. But because of invariance, $T \leq B$ has already been checked by rule in row 3, column 1. Therefore $B \cap T = T$. This is the reason the "forward" direction of the comparison needs to be checked first: otherwise, we would have updated B to equal T already and the $T \leq B$ comparison would become vacuous. Alternatively, we could actually compute $B \cap T$. However there is reason to suspect that trouble lies that way. We would need to either add intersection types to the system, or compute a meet without them. Either way, the algorithm would become much more complex and, judging by past results, likely undecidable.

Complexity

These subtyping rules are likely Π_2^{P}-hard. Checking a subtype relation with unions requires checking that for all choices on the left, there exists a choice on the right that makes the relation hold. This matches the quantifier structure of 2-TQBF problems of the form $\forall x_i \exists y_i F$ where F is a boolean formula. If the formula is written 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). We have not worked out the details, but this sketch is reasonably convincing. Π_2^{P} is only the most obvious reduction to try; it is possible our system equals PSPACE or even greater, as has often been the case for subtyping systems like ours.

Implementation

Appendix A gives a Julia implementation of this algorithm.

Example deductions

We will briefly demonstrate the power of this algorithm through examples of type relationships it can determine. In these examples, note that $<$ means less than but not equal. `Pair`

is assumed to be a tag type with two parameters.

The algorithm can determine that a type matches constraints specified by another:

$$\texttt{Tuple}\{\texttt{Array}\{\texttt{Integer}, 1\}, \texttt{Int}\} \ < \ (\exists\ T <: \texttt{Integer}\ \exists\ S <: T\ \texttt{Tuple}\{\texttt{Array}\{T, 1\}, S\})$$

It is not fooled by redundant type variables:

$$\texttt{Array}\{\texttt{Int}, 1\} \ = \ \texttt{Array}\{(\exists\ T <: \texttt{Int}\ T), 1\}$$

Variables can have non-trivial bounds that refer to other variables:

$$\begin{aligned}&\texttt{Pair}\{\texttt{Float32}, \texttt{Array}\{\texttt{Float32}, 1\}\} \ < \\ &\quad \exists\ T <: \texttt{Real}\ \exists\ S <: \texttt{AbstractArray}\{T, 1\}\ \texttt{Pair}\{T, S\}\end{aligned}$$

In general, if a variable appears multiple times then its enclosing type is more constrained than a type where variables appear only once:

$$\exists\ T\ \texttt{Pair}\{T, T\} \ < \ \exists\ T\ \exists\ S\ \texttt{Pair}\{T, S\}$$

However with sufficiently tight bounds that relationship no longer holds (here x is any type):

$$\exists\ x <: T <: x\ \exists\ x <: S <: x\ \texttt{Pair}\{T, S\} \ < \ \exists\ T\ \texttt{Pair}\{T, T\}$$

Variadic tuples of unions are particularly expressive ("every argument is either this or that"):

$$\begin{aligned}&\texttt{Tuple}\{\texttt{Vector}\{\texttt{Int}\},\texttt{Vector}\{\texttt{Vector}\{\texttt{Int}\}\},\texttt{Vector}\{\texttt{Int}\},\texttt{Vector}\{\texttt{Int}\}\} \ < \\ &\quad \exists\ S <: \texttt{Real}\ \texttt{Tuple}\{\texttt{Union}\{\texttt{Vector}\{S\},\texttt{Vector}\{\texttt{Vector}\{S\}\}\}\ldots\}\end{aligned}$$

And the algorithm understands that tuples distribute over unions:

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

Pseudo-contravariance

Bounded existential types are normally contravariant in their lower bounds. However, the set denotations of our types give us $(\exists\ T >: X\ T) = \texttt{Any}$. By interposing an invariant

constructor, we obtain a kind of contravariance:

$$(\exists\ T >: X\ \texttt{Ref}\{T\}) \le (\exists\ T >: Y\ \texttt{Ref}\{T\}) \implies Y \le X$$

Types like this may be useful for describing nominal arrow types (see section 4.6.3). This ability to swap types between sides of $\le$ is one of the keys to the undecidability of subtyping with bounded quantification [93]. Nevertheless we conjecture that our system is decidable, since we treat left- and right-side variables asymmetrically. The bounds of left-side variables do not change, which might be sufficient to prevent the "re-bounding" process in system $F_{<:}$ from continuing.

Related work

Our algorithm is similar to some that have been investigated for Java generics ([115, 20, 111]). In that context, the primary difficulty is undecidability due to circular constraints such as `class Infinite<P extends Infinite<?>>` and circularities in the inheritance hierarchy such as `class C implements List<List<? super C>>` (examples from [111]). We have not found a need for such complex declarations. Julia disallows mentioning a type in the constraints of its own parameters. When declaring type T a subtype of S, only the direct parameters of T or S can appear in the subtype declaration. We believe this leads to decidable subtyping.

Dynamic typing provides a useful loophole: it is never truly necessary to declare constraints on a type's parameters, since the compiler does not need to be able to prove that those parameters satisfy any constraints when instances of the type are used. In theory, an analogous problem for Julia would be spurious method ambiguities due to an inability to declare sufficiently strong constraints. More work is needed to survey a large set of Julia libraries to see if this might be a problem in practice.

Past work on subtyping with regular types [58, 49] is also related, particularly to our treatment of union types and variadic tuples.

4.3 Dispatch system

Julia's dispatch system strongly resembles the multimethod systems found in some object-oriented languages [15, 37, 103, 28, 29, 30]. However we prefer the term type-based dispatch, since our system actually works by dispatching a *single tuple type* of arguments. The difference is subtle and in many cases not noticeable, but has important conceptual implications. It means that methods are not necessarily restricted to specifying a type for each argument "slot". For example a method signature could be `Union{Tuple{Any,Int}, Tuple{Int,Any}}`, which matches calls where either, but not necessarily both, of two arguments is an `Int`. [2]

4.3.1 Type and method caches

The majority of types that occur in practice are *simple*. A simple type is a tag, tuple, or abstract type, all of whose parameters are simple types or non-type values. For example `Pair{Int,Float64}` is a simple type. Structural equality of simple types is equivalent to type equality, so simple types are easy to compare, hash, and sort. Another important set of types is the *concrete* types, which are the direct types of values. Concrete types are hash-consed in order to assign a unique integer identifier to each. These integer identifiers are used to look up methods efficiently in a hash table.

Some types are concrete but not simple, for example `Array{Union{Int,String},1}`. The hash-consing process uses linear search for these types. Fortunately, such types tend to make up less than 10% of the total type population. On cache misses, method tables also use linear search.

4.3.2 Specificity

Sorting methods by specificity is a crucial feature of a generic function system. It is not obvious how to order types by specificity, and our rules for it have evolved with experience. The core of the algorithm is straightforward:

[2] Our implementation of Julia does not yet have syntax for such methods.

- If A is a strict subtype of B, then it is more specific than B.
- If $B \leq A$ then A is not more specific than B.
- If some element of tuple type A is more specific than its corresponding element in tuple type B, and no element of B is more specific than its corresponding element in A, then A is more specific.

This is essentially the same specificity rule used by Dylan [103] for argument lists. Julia generalizes this to tuple types; it is applied recursively to tuple types wherever they occur. We then need several more rules to cover other features of the type system:

- A variadic type is less specific than an otherwise equal non-variadic type.
- Union type A is more specific than B if some element of A is more specific than B, and B is not more specific than any element of A.
- Non-union type A is more specific than union type B if it is more specific than some element of B.
- A tag type is more specific than a tag type strictly above it in the hierarchy, regardless of parameters.
- A variable is more specific than type T if its upper bound is more specific than T.
- Variable A is more specific than variable B if A's upper bound is more specific than B's, and A's lower bound is not more specific than B's.

So far, specificity is clearly a less formal notion than subtyping.

Specificity and method selection implicitly add a set difference operator to the type system. When a method is selected, all more specific definitions have not been, and therefore the more specific signatures have been subtracted. For example, consider the following definitions of a string concatenation function:

```
strcat(strs::ASCIIString...) = # an ASCIIString

strcat(strs::Union{ASCIIString,UTF8String}...) = # a UTF8String
```

Inside the second definition, we know that at least one argument must be a `UTF8String`, since otherwise the first definition would have been selected. This encodes the type behavior that ASCII strings are closed under concatenation, while introducing a single UTF-8 string requires the result to be UTF-8 (since UTF-8 can represent strictly more characters than ASCII). The effect is that we obtain some of the power of set intersection and negation without requiring subtyping to reason explicitly about such types.

4.3.3 Parametric dispatch

We use the following syntax to express methods whose signatures are `UnionAll` types:

```
func{T<:Integer}(x::Array{T}) = ...
```

This method matches any `Array` whose element type is some subtype of `Integer`. The value of `T` is available inside the function. This feature largely overcomes the restrictiveness of parametric invariance. Methods like these are similar to methods with "implicit type parameters" in the Cecil language [29]. However, Cecil (along with its successor, Diesel [30]) does not allow methods to differ only in type parameters. In the semantic subtyping framework we use, such definitions have a natural interpretation in terms of sets, and so are allowed. In fact, they are used extensively in Julia (see section 5.4 for an example).

4.3.4 Diagonal dispatch

`UnionAll` types can also express constraints between arguments. The following definition matches two arguments of the same concrete type:

```
func{T}(x::T, y::T) = ...
```

Section 5.1 discusses an application.

This feature is currently implemented only in the dispatch system; we have not yet formalized it as part of subtyping. For example, the subtyping algorithm presented here concludes that `Tuple{Int,String}` is a subtype of `UnionAll T Tuple{T,T}`, since `T` might equal `Any` or `Union{Int,String}`. We believe this hurts the accuracy of static type deductions, but not their soundness, as long as we are careful in the compiler to treat all types as over-estimates of run time types. In any case, we plan to fix this by allowing some type variables to be constrained to concrete types. This only affects type variables that occur only in covariant position; types such as `UnionAll T Tuple{Array{T},T}` are handled correctly.

4.3.5 Constructors

Generic function systems (such as Dylan's) often implement value constructors by allowing methods to be specialized for classes themselves, instead of just instances of classes. We extend this concept to our type system using a construct similar to singleton kinds [107]: `Type{T}` is the type of any type equal to `T`. A constructor for a complex number type can then be written as follows:

```
call{T}(::Type{Complex{T}}, re::Real, im::Real) =
    new(Complex{T}, re, im)
```

This requires a small adjustment to the evaluation rules: in the application syntax $e_1(e_2)$ when e_1 is not a function, evaluate `call`(e_1, e_2) instead, where `call` is a particular generic function known to the system. This constructor can then be invoked by writing e.g. `Complex{Float64}(x,y)`.

Naturally, `Type` can be combined arbitrarily with other types, adding a lot of flexibility to constructor definitions. Most importantly, type parameters can be omitted in constructor calls as long as there is a `call` method for the "unspecialized" version of a type:

```
call{T<:Real}(::Type{Complex}, re::T, im::T) =
    Complex{T}(re, im)
```

This definition allows the call `Complex(1.0,2.0)` to construct a `Complex{Float64}`. This mechanism subsumes and generalizes the ability to "infer" type parameters found in

some languages. It also makes it possible to add new parameters to types without changing client code.

4.3.6 Ambiguities

As in any generic function system, method ambiguities are possible. Two method signatures are ambiguous if neither is more specific than the other, but their intersection is non-empty. For arguments that are a subtype of this intersection, it is not clear which method should be called.

Some systems have avoided this problem by resolving method specificity left to right, i.e. giving more weight to the leftmost argument. With dispatch based on whole tuple types instead of sequences of types, this idea seems less defensible. One could extract a sequence of types by successively intersecting a method type with `UnionAll T Tuple{T,Any...}`, then `UnionAll T Tuple{Any,T,Any...}`, etc. and taking the inferred `T` values. However our experience leads us to believe that the left to right approach is a bit too arbitrary, so we have stuck with "symmetric" dispatch.

In the Julia library ecosystem, method ambiguities are unfortunately common. Library authors have been forced to add many tedious and redundant disambiguating definitions. Most ambiguities have been mere nuisances rather than serious problems. A representative example is libraries that add new array-like containers. Two such libraries are Images and DataArray (an array designed to hold statistical data, supporting missing values). Each library wants to be able to interoperate with other array-like containers, and so defines e.g. `+(::Image, ::AbstractArray)`. This definition is ambiguous with `+(::AbstractArray, ::DataArray)` for arguments of type `Tuple{Image, DataArray}`. However in practice Images and DataArrays are not used together, so worrying about this is a bit of a distraction. Giving a run time error forcing the programmer to clarify intent seems to be an acceptable solution (this is what Dylan does; currently Julia only prints a warning when the ambiguous definition is introduced).

4.4 Generic programming

Modern object-oriented languages often have special support for "generic" programming, which allows classes, methods, and functions to be reused for a wider variety of types. This capability is powerful, but has some usability cost as extra syntax and additional rules must be learned. We have found that the combination of our type system and generic functions subsumes many uses of generic programming features.

For example, consider this C++ snippet, which shows how a template parameter can be used to vary the arguments accepted by methods:

```
template <typename T>
class Foo<T> { int method1(T x); }
```

The method `method1` will only accept `int` when applied to a `Foo<int>`, and so on. This pattern can be expressed in Julia as follows:

```
method1{T}(this::Foo{T}, x::T) = ...
```

Associated types

When dealing with a particular type in a generic program, it is often necessary to mention other types that are somehow related. For example, given a collection type the programmer will want to refer to its element type, or given a numeric type one might want to know the next "larger" numeric type. Such associated types can be implemented as generic functions:

```
eltype{T}(::AbstractArray{T}) = T

widen(::Type{Int32}) = Int64
```

Simulating multiple inheritance

It turns out that external multiple dispatch is sufficiently powerful to simulate a kind of multiple inheritance. At a certain point in the development of Julia's standard library, it become apparent that there were object properties that it would be useful to dispatch on,

but that were not reflected in the type hierarchy. For example, the array type hierarchy distinguishes dense and sparse arrays, but not arrays whose memory layout is fully contiguous. This is important for algorithms that run faster using a single integer index instead of a tuple of indexes to reference array elements.

Tim Holy pointed out that new object properties can be added externally [56], and proposed using it for this purpose:

```
abstract LinearIndexing

immutable LinearFast <: LinearIndexing; end
immutable LinearSlow <: LinearIndexing; end

linearindexing(A::Array) = LinearFast()
```

This allows describing the linear indexing performance of a type by adding a method to `linearindexing`. An algorithm can consume this information as follows:

```
algorithm(a::AbstractArray) = _algorithm(a, linearindexing(a))

function _algorithm(a, ::LinearFast)
    # code assuming fast linear indexing
end

function _algorithm(a, ::LinearSlow)
    # code assuming slow linear indexing
end
```

The original author of a type like `Array` does not need to know about this property, unlike the situation with multiple inheritance or explicit interfaces. Adding such properties after the fact makes sense when new algorithms are developed whose performance is sensitive to distinctions not previously noticed.

4.5 Staged programming

An especially fruitful use of types in Julia is as input to code that generates other code, in a so-called "staged" program. This feature is accessed simply by annotating a method definition with a macro called `@generated`:

```
@generated function getindex{T,N}(a::Array{T,N}, I...)
    expr = 0
    for i = N:-1:1
        expr = :( (I[$i]-1) + size(a,$i)*$expr )
    end
    :(linear_getindex(a, $expr + 1))
end
```

Here the syntax `:( )` is a quotation: a representation of the expression inside is returned, instead of being evaluated. The syntax `quote ... end` is an alternative for quoting multi-line expressions. The value of an expression can be interpolated into a quotation using prefix `$`.

This simple example implements indexing with `N` arguments in terms of linear indexing, by generating a fully unrolled expression for the index of the element. Inside the body of the method, argument names refer to the types of arguments instead of their values. Parameter values `T` and `N` are available as usual. The method returns a quoted expression. This expression will be used, verbatim, as if it were the body of a normal method definition for these argument types.

The remarkable thing about this is how seamlessly it integrates into the language. The method is selected like any other; the only difference is that the method's expression-generating code is invoked on the argument types just before type inference. The optimized version of the user-generated code is then stored in the method cache, and can be selected in the future like any other specialization. The generated code can even be inlined into call sites in library clients. The caller does not need to know anything about this, and the author of the function does not need to do any bookkeeping.

One reason this works so well is that the amount of information available at this stage is well balanced. If, for example, only the *classes* of arguments in a traditional object-oriented language were available, there would not be much to base the generated code on. If, on the other hand, details of the contents of values were needed, then it would be difficult to reuse the existing dispatch system.

The types of arguments say something about their *meaning*, and so unlike syntax-based

systems (macros) are independent of how the arguments are computed. A staged method of this kind is guaranteed to produce the same results for arguments `2+2` and `4`.

Another advantage of this approach is that the expression-generating code is purely functional. Staged programming with `eval`, in contrast, requires arbitrary user code to be executed in all stages. This has its uses; one example mentioned in [39] is reading a schema from a database in order to generate code for it. But with `@generated`, code is a pure function of type information, so when possible it can be invoked once at compile time and then never again. This preserves the possibility of full static compilation, but due to dynamic typing the language cannot guarantee that all code generation will occur before run time.

Trade-offs of staged programming

Code that generates code is more difficult to write, read, and debug. At the same time, staged programming is easy to overuse, leading to obscure programs that could have been written with basic language constructs instead.

This feature also makes static compilation more difficult. If an unanticipated call to a staged method occurs in a statically compiled program, two rather poor options are available: stop with an error, or run the generated code in an interpreter (we could compile code at run time, but presumably the purpose of static compilation was to avoid that). Ordinarily, the worst case for performance is to dynamically dispatch every call. Constructing expressions and evaluating them with an interpreter is even worse, especially considering that performance is the only reason to use staged programming in the first place. This issue will need to be addressed through enhanced static analysis and error reporting tools.

It would be nice to be able to perform staging based on any types, including abstract types. This would provide a way to generate fewer specializations, and could aid static compilation (by generating code for the most general type `Tuple{Any...}`). Unfortunately this appears to be impractical, due to the need for functions to be monotonic in the type domain. The soundness and termination of data flow analysis depend on the property that given lattice elements a and b, we must have $a \subseteq b \implies f(a) \subseteq f(b)$ for all f. Since staged

methods can generate arbitrarily different code for different types, it is easy to violate this property. For the time being, we are forced to invoke staging only on concrete types.

Related work

Our approach is closely related to Exotypes [39], and to generation of high performance code using lightweight modular staging (LMS) [99, 92]. The primary difference is that in our case code generation is invoked implicitly, and integrated directly with the method cache. Unlike LMS, we do not do binding time analysis, so expression generation itself is more "manual" in our system.

4.6 Higher-order programming

Generic functions are first-class objects, and so can be passed as arguments just as in any dynamically typed language with first-class functions. However, assigning useful type tags to generic functions and deciding how they should dispatch is not so simple. Past work has often described the types of generic functions using the "intersection of arrows" formalism [100, 41, 22, 24]. Since an ordinary function has an arrow type $A \to B$ describing how it maps arguments A to results B, a function with multiple definitions can naturally be considered to have multiple such types. For example, a `sin` function with the following two definitions:

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

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

This approach is effective for statically checking uses of generic functions: anywhere a function goes, we must keep track of which arrow types it "contains" in order to be sure

that at least one matches every call site and allows the surrounding code to type check. However, despite the naturalness of this typing of generic functions, this formulation is quite problematic for dispatch and code specialization (not to mention that it might make subtyping undecidable).

4.6.1 Problems for code selection

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

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

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

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

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

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

A slightly different approach we might try would be to imitate the types of higher-

order functions in traditional statically typed functional languages. Consider the classic `map` function, which creates a new container by calling a given function on every element of a given container. This is a general pattern that occurs in "vectorized" functions in technical computing environments, e.g. when a function like `+` or `sin` operates on arrays elementwise. We might wish to write `map` as follows:

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

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

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

4.6.2 Problems for code specialization

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

For example, say we have a definition `f(g::String->String)`, and a function `h` with a single `Int` $\rightarrow$ `Int` definition. Naturally, `f` is not applicable to `h`. However, given the

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

Table 4.1: Number and percentage of pairs of functions with matching arguments, or matching arguments and return types. The second column gives the number of functions that have matches. The third column gives the mean number of matches per function.

call site `f(h)`, we are forced to conclude that `f` might be called with a function of type $(\texttt{Int} \to \texttt{Int}) \cap (\texttt{String} \to \texttt{String})$, since in general $\texttt{Int} \to \texttt{Int}$ might be only an approximation of the true type of the argument.

The other major concern when specializing code is whether, having generated code for a certain type, we would be able to reuse that code often enough for the effort to be worthwhile. In the case of arrow types, this equates to asking how often generic functions share the same set of signatures. This question can be answered empirically. Studying the Julia `Base` library as of this writing, there are 1059 generic functions. We examined all 560211 pairs of functions; summary statistics are shown in table 4.1. Overall, it is rare for functions to share type signatures. Many of the 85 functions with matches (meaning there exists some other function with the same type) are predicates, which all have types similar to $\texttt{Any} \to \texttt{Bool}$. The mean of 0.23 means that if we pick a function uniformly at random, on average 0.23 other functions will match it. The return types compared here depend on our heuristic type inference algorithm, so it useful to exclude them in order to get an upper bound. If we do that, and only consider arguments, the mean number of matches rises to 1.73.

The specific example of the `sin` and `cos` functions provides some intuition for why there are so few matches. One would guess that the type behavior of these functions would be identical, however the above evaluation showed this not to be the case. The reason is that the functions have definitions to make them operate elementwise on both dense and sparse arrays. `sin` maps zero to zero, but `cos` maps zero to one, so `sin` of a sparse array gives a sparse array, but `cos` of a sparse array gives a dense array. This is indicative of the general

"messiness" of convenient real-world libraries for technical computing.

4.6.3 Possible solutions

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

Another approach that is especially relevant to technical computing is to use nominal function types. In mathematics, the argument and return types of a function are often not its most interesting properties. In some domains, for example, all functions can implicitly be assumed $\mathbb{R} \to \mathbb{R}$, and the interesting property might be what order of integrable singularity is present (see section 5.7 for an application), or what dimension of linear operator the function represents. The idea of nominal function types is to describe the properties of interest using a data object, and then allow that data object to be treated as a function, i.e. "called". Some object-oriented languages call such an object a *functor*.

Julia accommodates this approach with the `call` mechanism used for constructors (see section 4.3.5).

As an example, we can define a type for polynomials of order N over ring R:

```
immutable Polynomial{N,R}
    coeffs::Vector{R}
end

function call{N}(p::Polynomial{N}, x)
    v = p.coeffs[end]
    for i = N:-1:1
        v = v*x + p.coeffs[i]
    end
    return v
```

```
end
```

Now it is possible to use a `Polynomial` just like a function, while also dispatching methods on the relevant properties of order and ring (or ignoring them if you prefer). It is also easy to examine the polynomial's coefficients, in contrast to functions, which are usually opaque.

Another possible use of this feature is to implement automatic vectorization of scalar functions, as found in Chapel [27]. Only one definition would be needed to lift any declared subtype of `ScalarFunction` to arrays:

```
call(f::ScalarFunction, a::AbstractArray) = map(f, a)
```

It is even possible to define a "nominal arrow" type, which uses this mechanism to impose a classification on functions based on argument and return types:

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

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

Calling an `Arrow` will yield a no-method error if the argument type fails to match, and a type error if the return type fails to match.

4.6.4 Implementing `map`

So, given typed containers and no arrow types, how do you implement `map`? The answer is that the type of container to return must also be computed. This should not be too surprising, since it is implied by the definition of `new` at the beginning of the chapter: each value is created by computing a type part and a content part.

A possible implementation of `map` for arrays is shown in figure 4-2. The basic strategy is to try calling the argument function `f` first, see what it returns, and then optimistically assume that it will always return values of the same type. This assumption is checked on each iteration. If `f` returns a value that does not fit in the current output array `dest`, we re-allocate the output array to a larger type and continue. The primitive `typejoin` computes

```
function map(f, A::Array)
    isempty(A) && return Array(Bottom, 0)
    el = f(A[1]); T = typeof(el)
    dest = Array(T, length(A))
    dest[1] = el
    for i = 2:length(A)
        el = f(A[i]); S = typeof(el)
        if !(S <: T)
            T = typejoin(T, S)
            new = similar(dest, T)
            copy!(new, 1, dest, 1, i-1)
            dest = new
        end
        dest[i] = el::T
    end
    return dest
end
```

Figure 4-2: A Julia implementation of `map`. The result type depends only on the actual values computed, made efficient using an optimistic assumption.

a union-free join of types. For uses like `map`, this does not need to be a true least upper bound; any reasonable upper bound will do.

This implementation works well because it completely ignores the question of whether the compiler understands the type behavior of `f`. However, it *cooperates* with the compiler by making the optimistic assumption that `f` is well-behaved. This is best illuminated by considering three cases. In the first case, imagine `f` always returns `Int`, and that the compiler is able to infer that fact. Then the test `!(S <: T)` disappears by specialization, and the code reduces to the same simple and efficient loop we might write by hand. In the second case, imagine `f` always returns `Int`, but that the compiler is *not* able to infer this. Then we incur the cost of an extra type check on each iteration, but we return the same efficiently stored `Int`-specialized `Array` (`Array{Int}`). This leads to significant memory savings, and allows subsequent operations on the returned array to be more efficient. The third case occurs when `f` returns objects of different types. Then the code does not do anything particularly efficient, but is not much worse than a typical dynamically typed language manipulating

heterogeneous arrays.

Overall, the resulting behavior is quite similar to a dynamic language that uses storage strategies [17] for its collections. The main difference is that the behavior is implemented at the library level, rather than inside the runtime system. This can be a good thing, since one might want a `map` that works differently. For example, replacing `typejoin` with `promote_type` would collect results of different numeric types into a homogeneous array of a single larger numeric type. Other applications might not want to use typed arrays at all, in which case `map` can be much simpler and always return an `Array{Any}`. Still other applications might want to arbitrarily mutate the result of `map`, in which case it is difficult for any automated process to predict what type of array is wanted.

It must be admitted that this `map` is more complex than what we are used to in typical functional languages. However, this is at least partly due to `map` itself being more complex, and amenable to more different interpretations, than what is usually considered in the context of those languages.

4.7 Performance model

4.7.1 Type inference

Our compiler performs data flow type inference [62, 35] using a graph-free algorithm [87]. Type inference is invoked on a method cache miss, and recursively follows the call graph emanating from the analyzed function. The first non-trivial function examined tends to cause a large fraction of loaded code to be processed.

The algorithmic components needed for this process to work are subtyping (already covered), join (which can be computed by forming a union type), meet ($\sqcap$), widening operators, and type domain implementations (also known as transfer functions) of core language constructs.

Meets are challenging to compute, and we do not yet have an algorithm for it that is as rigorously developed as our subtyping. Our current implementation traverses two types,

recursively applying the rules

$$\begin{aligned} T \le S &\implies T \sqcap S = T \\ S \le T &\implies T \sqcap S = S \\ \mathit{otherwise} &\quad T \sqcap S = \bot \end{aligned}$$

As this is done, constraints on all variables are accumulated, and then solved. $(A \cup B) \sqcap C$ is computed as $(A \sqcap C) \cup (B \sqcap C)$. Due to the way meet is used, it is safe for it to over estimate its result. That will only cause us to conclude that more methods are applicable than really are, leading to coarser but still sound type inference. Note that for concrete type T, the rule $T \le S \implies T \sqcap S = T$ suffices.

Widening is needed in three places:

- Function argument lists that grow in length moving down the call stack.
- Computing the type of a field. Instantiating field types builds up other types, potentially leading to infinite chains.
- When types from two control flow paths are merged.

Each kind of type has a corresponding form of widening. A type nested too deeply can be replaced by a `UnionAll` type with variables replacing each component that extends beyond a certain fixed depth. A union type can be widened using a union-free join of its components. A long tuple type is widened to a variadic tuple type.

Since the language has few core constructs, not many transfer functions are needed, but the type system requires them to be fairly sophisticated. For example accessing an unknown index of `Tuple{Int...}` can still conclude that the result type is `Int`. The most important rule is the one for generic functions:

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

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

4.7.2 Specialization

It might be thought that Julia is fast because our compiler is well engineered, perhaps using "every trick in the book". At risk of being self-deprecating, this is not the case. After type inference we apply standard optimizations: static method resolution, inlining, specializing variable storage, eliminating tuples that do not escape local scope, unboxing function arguments, and replacing known function calls with equivalent instruction sequences.

The performance we manage to get derives from language design. Writing complex functions with many behaviors as generic functions naturally encourages them to be written in a style that is easier to analyze statically.

Julia specializes methods for nearly every combination of concrete argument types that occurs. This is undeniably expensive, and past work has often sought to avoid or limit specialization (e.g. [40]).

4.7.3 Performance predictability

Performance predictability is one of the main sacrifices of our design. There are three traps: widening (which is heuristic), changing the type of a variable, and heterogeneous containers (e.g. arrays of `Any`). These can cause significant slowdowns for reasons that may be unclear to the programmer. To address this, we provide a sampling profiler as well as the ability to examine the results of type inference. The package `TypeCheck.jl` [55] can also be used to provide warnings about poorly-typed code.

4.8 Dispatch utilization

To evaluate how dispatch is actually used in our application domain, we applied the following metrics [91]:

1. Dispatch ratio (DR): The average number of methods in a generic function.

Language	DR	CR	DoS
Gwydion	1.74	18.27	2.14
OpenDylan	2.51	43.84	1.23
CMUCL	2.03	6.34	1.17
SBCL	2.37	26.57	1.11
McCLIM	2.32	15.43	1.17
Vortex	2.33	63.30	1.06
Whirlwind	2.07	31.65	0.71
NiceC	1.36	3.46	0.33
LocStack	1.50	8.92	1.02
Julia	5.86	51.44	1.54
Julia Operators	28.13	78.06	2.01

Table 4.2: Comparison of Julia (1208 functions exported from the `Base` library) to other languages with multiple dispatch, based on dispatch ratio (DR), choice ratio (CR), and degree of specialization (DoS) [91]. The "Julia Operators" row describes 47 functions with special syntax (binary operators, indexing, and concatenation).

2. Choice ratio (CR): For each method, the total number of methods over all generic functions it belongs to, averaged over all methods. This is essentially the sum of the squares of the number of methods in each generic function, divided by the total number of methods. The intent of this statistic is to give more weight to functions with a large number of methods.

3. Degree of specialization (DoS): The average number of type-specialized arguments per method.

Table 4.2 shows the mean of each metric over the entire Julia `Base` library, showing a high degree of multiple dispatch compared with corpora in other languages [91]. Compared to most multiple dispatch systems, Julia functions tend to have a large number of definitions.

To see why this might be, it helps to compare results from a biased sample of only operators. These functions are the most obvious candidates for multiple dispatch, and as a result their statistics climb dramatically. Julia has an unusually large proportion of functions with this character.

Chapter 5

Case studies

This chapter discusses several examples that illustrate the effectiveness of Julia's abstractions for technical computing. The first three sections provide an "explication through elimination" of core features (numbers and arrays) that have usually been built in to technical computing environments. The remaining sections introduce some libraries and real-world applications that benefit from Julia's design.

5.1 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 don't want to write all $O(n^2)$ possibilities. We need abstract definitions that cover many points in the dispatch matrix. One such family of points is particularly important: those that describe converting a value to a type it is already an instance of. In our system this can be handled by a single definition that performs "triangular" dispatch:

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

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

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

As with conversion, we would like to provide default definitions for == that cover families of cases. Numbers are a reasonable domain to pick, since all numbers should be equality-comparable to each other. We might try (`Number` is the abstract supertype of all numeric types):

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

meaning that number comparison can simply fall back to intensional equality. However this definition is rather dubious. It gets the wrong answer every time the arguments are different representations (e.g. integer and floating point) of the same quantity. We might hope that its behavior will be "patched up" later by more specific definitions for various concrete number types, but it still covers a dangerous amount of dispatch space. If later definitions somehow

miss a particular combination of number types, we could get a silent wrong answer instead of an error. (Note that statically checking method exhaustiveness is no help here.)

"Diagonal" dispatch lets us improve the definition:

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

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

5.2 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: each new set of numbers has a subset isomorphic to an existing set. 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 [34]: 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").

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 (the situation is of course improved in C++). In Python, a special arrangement is made for + to call either an `__add__` or `__radd__` method, effectively providing double-dispatch for arithmetic in a language that is idiomatically single-dispatch.

Implementing type embeddings

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

It has been suggested on theoretical grounds [98] that generic binary operators should have "key variants" where the arguments are of the same type. We implement this in Julia with a default definition that uses diagonal dispatch:

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

Then we can implement a more general definition for different-type arguments that tries to promote the arguments to a common type by calling `promote`:

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

`promote` returns a pair of the values of its arguments after conversion to a common type, so that a "key variant" can be invoked.

`promote` is designed to be extended by new numeric types. A full-featured promotion operator is a tall order. We would like

- Each combination of types only needs to be defined in one order; we don't want to redundantly define the behavior of `(T,S)` and `(S,T)`.

- It falls back to join for types without any defined promotion.
- It must prevent promotion above a certain point to avoid circularity.

The core of the mechanism is three functions: `promote_type`, which performs promotion in the type domain only, `promote_rule`, which is the function defined by libraries, and a utility function `promote_result`. The default definition of `promote_rule` returns `Bottom`, indicating no promotion is defined. `promote_type` calls `promote_rule` with its arguments in both possible orders. If one order is defined to give `X` and the other is not defined, `promote_result` recursively promotes `X` and `Bottom`, giving `X`. If neither order is defined, the last definition below is called, which falls back to `typejoin`:

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

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

promote_rule(T, S) = Bottom

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

The obvious way to extend this system is to define `promote_rule`, but it can also be extended in a less obvious way. For the purpose of manipulating container types, we would like e.g. `Int` and `Real` to promote to `Real`. However, we do not want to introduce a rule that promotes arbitrary pairs of numeric types to `Number`, since that would make the above default definition of `+` circular. The following definitions accomplish this, by promoting to the larger numeric type if one is a subtype of the other:

```
promote_result{T<:Number,S<:Number}(::Type{T}, ::Type{S},
                                    ::Type{Bottom}, ::Type{Bottom}) =
    promote_sup(T, S, typejoin(T,S))

# promote numeric types T and S to typejoin(T,S) if T<:S or S<:T
```

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

Application to ranges

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

On implicit conversion

In order to facilitate human styles of thinking, many programming languages offer some flexibility in when types are required to match. For example, a C function declared to accept type `float` can also be called with an `int`, and the compiler will insert a conversion automatically. We have found that most cases where this behavior is desirable are handled simply by method applicability, as shown in this and the previous section. Most of the remaining cases involve assignment. Julia has a notion of a *typed location*, which is a variable or mutable object field with an associated type. The core language requires types to match (as determined by subtyping) on assignment. However, updating assignments that are syntactically identifiable (via use of `=`) cause the front-end to insert calls to `convert` to try to convert the assignment right-hand side to the type of the assigned location. Abstract data types can easily mimic this behavior by (optionally) calling `convert` in their implementations of mutating operations. In our experience, this arrangement provides the needed convenience without complicating the language. Program analyses consuming lowered syntax trees (or any lower level representation) do not need to know anything about conversion or coercion.

Fortress [4] also provides powerful type conversion machinery. However, it does so through a special `coerce` feature. This allows conversions to happen during dispatch, so a method can become applicable if arguments can be converted to its declared type. There is also a built-in notion of type widening. This allows conversions to be inserted between nested operator applications, which can improve the accuracy of mixed-precision code. It is not possible to provide these features in Julia without doing violence to our "everything is a method call" semantics. We feel that it is worth giving up these features for a simpler mental model. We also believe that the ability to provide so much "magic" numeric type behavior only through method definitions is compelling.

Number-like types in practice

Originally, our reasons for implementing all numeric types at the library level were not entirely practical. We had a principled opposition to including such definitions in a compiler,

and guessed that being able to define numeric types would help ensure the language was powerful enough. However, defining numeric and number-like types and their interactions turns out to be surprisingly useful. Once such types are easy to obtain, people find more and more uses for them.

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

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

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

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

A problem in the design of image processing libraries was solved by defining a new kind of fixed-point type [63]. The problem is that image scientists often want to work with fractional pixel values in the interval $[0, 1]$, but most graphics libraries (and memory efficiency concerns) require 8-bit integer pixel components with values in the interval $[0, 255]$. The solution is a `Ufixed8` type that uses an unsigned 8-bit integer as its representation, but behaves like a fraction over 255.

Many real-world quantities are not numbers exactly, but benefit from the same mechanisms in their implementation. Examples include colors (which form a vector space, and

where many different useful bases have been standardized), physical units, and DNA nucleotides. Date and time arithmetic is especially intricate and irregular, and benefits from permissiveness in operator definitions.

5.3 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:

1. 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.

2. In some code the number of dimensions is a *dynamic* property — it is only known at run time.

3. 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, when computing 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 can 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 trade-off is made, rules must be defined for how various operators act on dimensions. Here we 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). A significant amount of work has been done on inferring properties like these for existing array-based languages (e.g. [61, 52], although these methods go farther and attempt to infer complete size information).

C++ and Haskell are both examples of languages with sufficiently powerful static semantics to support defining efficient and reasonably flexible multi-dimensional arrays in libraries. In both languages, implementing these libraries is fairly difficult and places some limits on how indexing can behave. In C++, the popular Boost libraries include a multi-array type [51]. To index an array with this library, one builds up an `index_gen` object by repeatedly applying the `[]` operator. On each application, if the argument is a `range` then the corresponding dimension is kept, and otherwise it is dropped. This is implemented with a template as follows:

```
template <int NumRanges, int NumDims>
struct index_gen {
index_gen<NumRanges+1,NumDims+1> operator[](const range& r) const
    { ... }

index_gen<NumRanges+1,NumDims> operator[](index idx) const
    { ... }
}
```

The number of dimensions in the result is determined by arithmetic on template parameters and static overloading. Handling arguments one at a time recursively is a common pattern

for feeding more complex type information to compilers. In fact, the Haskell library Repa [65] uses essentially the same approach:

```
instance Slice sl => Slice (sl :. Int) where
    sliceOfFull (fsl :. _) (ssl :. _) = sliceOfFull fsl ssl

instance Slice sl => Slice (sl :. All) where
    sliceOfFull (fsl :. All) (ssl :. s) = sliceOfFull fsl ssl :. s
```

The first instance declaration says that given an `Int` index, the dimension corresponding to `_` is dropped. The second declaration says that given an `All` index, dimension `s` is kept.

Our dispatch mechanism permits a novel solution [14]. Indexing behavior can be defined with method signatures, using a combination of variadic methods and argument "splatting" (the ability to pass a structure of n values as n separate arguments to a function, known as `apply` in Lisp dialects, and written as `f(xs...)` in Julia). This solution is still a compromise among the factors outlined above, but it is a new compromise that provides a modest increment of power.

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

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

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

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

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

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

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

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

5.4 Numerical linear algebra

The crucial roles of code selection and code specialization in technical computing are captured well in the linear algebra software engineer's dilemma, described in the context of LAPACK and ScaLAPACK by Demmel and Dongarra, *et al.* [38]:

```
(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)
```

Organizing this functionality has been a challenge. LAPACK is associated with dense matrices, but in reality supports 28 matrix data types (e.g. diagonal, tridiagonal, symmetric, symmetric positive definite, triangular, triangular packed, and other combinations). Considerable gains in performance and accuracy are possible by exploiting these structured cases. LAPACK exposes these data types by assigning each a 2-letter code used to prefix function names. This approach, while lacking somewhat in abstraction, has the highly redeeming

Structured matrix types	Factorization types
Bidiagonal	Cholesky
Diagonal	CholeskyPivoted
SymmetricRFP	QR
TriangularRFP	QRCompactWY
Symmetric	QRPivoted
Hermitian	Hessenberg
AbstractTriangular	Eigen
LowerTriangular	GeneralizedEigen
UnitLowerTriangular	SVD
UpperTriangular	Schur
UnitUpperTriangular	GeneralizedSchur
SymTridiagonal	LDLt
Tridiagonal	LU
UniformScaling	CholeskyDenseRFP

Table 5.1: Left column: structured array storage formats currently defined in Julia's standard library. Right column: data types representing the results of various matrix factorizations.

feature of making LAPACK easy to call from almost any language. Still there is a usability cost, and an abstraction layer is needed.

Table 5.1 lists some types that have been defined in Julia (largely by Andreas Jensen) for working with structured matrices and matrix factorizations. The structured matrix types can generally be used anywhere a 2-d array is expected. The factorization types are generally used for efficiently re-solving systems. Sections 2.2 and 3.1 alluded to some of the ways that these types interact with language features. There are essentially two modes of use: "manual", where a programmer explicitly constructs one of these types, and "automatic", where a library returns one of them to user code written only in terms of generic functions.

Structured matrices have their own algebra. There are embedding relationships like those discussed in section 5.2: diagonal is embedded in bidiagonal, which is embedded in tridiagonal. Bidiagonal is also embedded in triangular. After promotion, these types are closed under + and −, but generally not under ∗.

Calling BLAS and LAPACK

Writing interfaces to BLAS [69] and LAPACK is an avowed tradition in technical computing language implementation. Julia's type system is capable of describing the cases where routines from these libraries are applicable. For example the axpy operation in BLAS (computing $y \leftarrow \alpha x + y$) supports arrays with a single stride, and four numeric types. In Julia this can be expressed as follows:

```
const BlasFloat = Union(Float64,Float32,Complex128,Complex64)

axpy!{T<:BlasFloat}(alpha::Number,
                    x::Union(DenseArray{T},StridedVector{T}),
                    y::Union(DenseArray{T},StridedVector{T}))
```

5.5 Units

Despite their enormous importance in science, unit quantities have not reached widespread use in programming. This is not surprising considering the technical difficulties involved. Units are symbolic objects, so attaching them to numbers can bring significant overhead. To restore peak performance, units need to be "lifted" into a type system of some kind, to move their overhead to compile time. However at that point we encounter a trade-off similar to that present in multi-dimensional arrays. Will it be possible to have an array of numbers with different units? What if a program wants to return different units based on criteria the compiler cannot resolve? Julia's automatic blending of binding times is again helpful.

The SIUnits package by Keno Fischer [44] implements Julia types for SI units and unit quantities:

```
immutable SIUnit{m,kg,s,A,K,mol,cd} <: Number
end

immutable SIQuantity{T<:Number,m,kg,s,A,K,mol,cd} <: Number
  val::T
end
```

This implementation uses a type parameter to store the exponent (an integer, or perhaps in the near future a rational number) of each base unit associated with a value. An `SIUnit` has only the symbolic part, and can be used to mention units without committing to a representation. Its size, as a data type, is zero. An `SIQuantity` contains the same symbolic information, but wraps a number used to represent the scalar part of the unit quantity. Definitions like the following are used to provide convenient names for units:

```
  const Meter     = SIUnit{1,0,0,0,0,0,0}()
  const KiloGram  = SIUnit{0,1,0,0,0,0,0}()
```

The approach described in section 5.2 is used to combine numbers with units. Arithmetic is implemented as follows:

```
function +{T,S,m,kg,s,A,K,mol,cd}(x::SIQuantity{T,m,kg,s,A,K,mol,cd},
                                  y::SIQuantity{S,m,kg,s,A,K,mol,cd})
  val = x.val+y.val
  SIQuantity{typeof(val),m,kg,s,A,K,mol,cd}(val)
end

function +(x::SIQuantity, y::SIQuantity)
  error("Unit mismatch.")
end
```

In the first definition, the representation types of the two arguments do not have to match, but the units do (checked via diagonal dispatch). Any combination of `SIQuantity`s that is not otherwise implemented is a unit mismatch error.

When storing unit quantities in arrays, the array constructor in Julia's standard library is able to automatically select an appropriate type. If all elements have the same units, the symbolic unit information is stored only once in the array's type tag, and the array data uses a compact representation:

```
julia> a = [1m,2m,3m]
3-element Array{SIQuantity{Int64,1,0,0,0,0,0,0},1}:
 1 m
 2 m
 3 m
```

```
julia> reinterpret(Int,a)
3-element Array{Int64,1}:
 1
 2
 3
```

If different units are present, a wider type is chosen via the array constructor invoking `promote_type`:

```
julia> a = [1m,2s]
2-element Array{SIQuantity{Int64,m,kg,s,A,K,mol,cd},1}:
 1 m
 2 s
```

Unit quantities are different from most number types in that they are not closed under multiplication. Nevertheless, generic functions behave as expected. Consider a generic `prod` function that multiplies elements of a collection. With units, its result type can depend on the size of the collection:

```
julia> prod([2m,3m]), typeof(prod([2m,3m]))
(6 m^2, SIQuantity{Int64,2,0,0,0,0,0,0})

julia> prod([2m,3m,4m]), typeof(prod([2m,3m,4m]))
(24 m^3, SIQuantity{Int64,3,0,0,0,0,0,0})
```

For simple uses of units, Julia's compiler will generally be able to infer result types. For more complex cases like `prod` it may not be able to, but this will only result in a loss of performance. This is the trade-off a Julia programmer accepts.

5.6 Algebraic modeling with JuMP

An entire subfield of technical computing is devoted to solving optimization problems. Linear programming, the problem of finding variable values that maximize a linear function subject to linear constraints, is especially important and widely used, for example in operations research.

Real-world problem instances can be quite large, requiring many variables and constraints. Simply writing down a large linear program is challenging enough to require automation. Specialized languages known as Algebraic Modeling Languages (AMLs) have been developed for this purpose. One popular AML is AMPL [45], which allows users to specify problems using a high-level mathematical syntax. Here is an example model of a "knapsack" problem (from [76]) in AMPL syntax:

```
var x{j in 1..N} >= 0.0, <= 1.0;

maximize Obj:
  sum {j in 1..N} profit[j] * x[j];

subject to CapacityCon:
  sum {j in 1..N} weight[j] * x[j] <= capacity;
```

This syntax is "compiled" to a numeric matrix representation that can be consumed by standard solver libraries. Although solving the problem can take a significant amount of time, building the problem representation is often the bottleneck. AMPL is highly specialized to this task and offers good performance.

Providing the same level of performance and convenience for linear programming within a general purpose language has proven difficult. However many users would prefer this, since it would make it easier to provide data to the model and use the results within a larger system. Using Julia, Miles Lubin and Iain Dunning solved this problem, creating the first embedded AML, JuMP [76], to match both the performance and notational advantages of specialized tools.

The solution is a multi-stage program: first the input syntax is converted to conventional loops and function calls with a macro, then the types of arguments are used to decide how to update the model, and finally code specialized for the structure of the input problem runs. The second stage is handled by a combination of generic functions and Julia's standard specialization process. Model parameters can refer to variables in the surrounding Julia program, without JuMP needing to understand the entire context.

Lubin and Dunning provide an example of how this works. The input

```
@addConstraint(m, sumweight[j]*x[j], j=1:N + s == capacity)
```

is lowered to (lightly edited):

```
aff = AffExpr()
for i = 1:N
  addToExpression(aff, 1.0*weight[i], x[i])
end
addToExpression(aff, 1.0, s)
addToExpression(aff, -1.0, capacity)
addConstraint(m, Constraint(aff,"=="))
```

`addToExpression` includes the following methods (plus others):

```
addToExpression(ex::Number,   c::Number,   x::Number)
addToExpression(ex::Number,   c::Number,   x::Variable)
addToExpression(ex::Number,   c::Number,   x::AffExpr)
addToExpression(ex::Number,   c::Variable, x::Variable)
addToExpression(ex::Number,   c::AffExpr,  x::AffExpr)
addToExpression(aff::AffExpr, c::Number,   x::Number)
addToExpression(aff::AffExpr, c::Number,   x::Variable)
addToExpression(aff::AffExpr, c::Variable, x::Variable)
addToExpression(aff::AffExpr, c::Number,   x::AffExpr)
addToExpression(aff::AffExpr, c::Number,   x::QuadExpr)
```

When the arguments are all numbers, `ex + c*x` is computed directly. Or, given an `AffExpr` and a variable, the `AffExpr`'s lists of variables and coefficients are extended. Given two variables, a quadratic term is added. With this structure, new kinds of terms can be added with minimal code, without needing to update the more complicated syntax transformation code.

Table 5.2 shows performance results. JuMP is only marginally slower than AMPL or direct use of the Gurobi solver's C++ API. The C++ implementation, unlike the other systems compared in the table, is not solver-independent and does not provide convenient high-level syntax. PuLP [86] is Python-based, and was benchmarked under the PyPy JIT compiler [16].

L	**JuMP/Julia**	**AMPL**	**Gurobi/C++**	**PuLP/PyPy**
1000	1.0	0.7	0.8	4.8
5000	4.2	3.3	3.9	26.4
10000	8.9	6.7	8.3	50.6
50000	48.3	35.0	39.3	225.6

Table 5.2: Performance (time in seconds) of several linear programming tools for generating models in MPS format (excerpted from [76]). L is the number of locations solved for in a facility location problem.

5.7 Boundary element method

There are lots of general packages [74, 95] implementing the finite-element method (FEM) [117] to solve partial differential equations (PDEs), but it is much more difficult to create such a "multi-physics" package for the boundary-element method (BEM) [18, 33] to solve surface-integral equations (SIEs). One reason is that BEM requires integrating functions with singularities many times in the inner loop of the code that builds the problem matrix, and doing this efficiently requires integration code, typically written by hand, that is specialized to the specific problem being solved.

Recent work [97] managed a more general solution using Mathematica to generate C++ code for different cases, in combination with some hand-written code. This worked well, but was difficult to implement and the resulting system is difficult to apply to new problems.. 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. Instead, we believe that the entire problem can be solved efficiently in Julia, relying on Julia's rich dispatch system and integrated code-generation facilities, and especially on staged functions to automate the generation of singular-integration code while retaining the interface of a simple function

library.

Thanks to Professor Steven Johnson for providing the following section describing the problem.

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[19, 18, 117], 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).

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

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

- Integral operators typically arise for *surface* integral equations (SIEs) [18, 33], and involve unknowns on a surface. The analogue of the FEM discretization is then a boundary element method (BEM) [18, 33], 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 and similar schemes [32, 72], 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 [73] 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.

Novel solution using Julia

Julia's dispatch and specialization features make this problem significantly easier to address:

- Dispatch on structured types allows the cubature scheme to be selected 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 run time 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 [97] have provided efficient cubature-generation algorithms of this sort, but it has not yet been practical to integrate them with run time code in a completely automated way.

A prototype implementation of this approach follows. First we define nominal function types that represent kernel functions:

```
abstract AbstractKernel

# any kernel that decays as X^p for X ≪ s and X^q for X ≫ s
abstract APowerLaw{p,q,s} <: AbstractKernel

# r^p power law
type PowerLaw{p} <: APowerLaw{p,p}; end
```

Next we add a type representing the integral $\mathcal{K}_n(X) = \int_0^1 w^n K(wX)dw$, and implement it for the case $K(x) = x^p$:

```
type FirstIntegral{K<:AbstractKernel,n}; end

function call{p,n}(::FirstIntegral{PowerLaw{p},n}, X::Number)
    return p >= 0 ? X^p / (1 + n + p) : inv(X^(-p) * (1 + n + p))
end
```

Code for instances of this function is specialized for p and n. Here is a sample session creating a function instance, and showing the LLVM [68] code generated for it:

```
F = FirstIntegral{PowerLaw{-1}, 3}()

@code_llvm F(2.5)

define double @julia_call_90703(%jl_value_t*, double) {
top:
  %2 = fmul double %1, 3.000000e+00
  %3 = fdiv double 1.000000e+00, %2
  ret double %3
}
```

Analytically known special cases can be added for other kernels. Here are two more. Notice that a Helmholtz kernel is also a power law kernel [1]:

```
type Helmholtz{k} <: APowerLaw{-1,-1}; end  # e^ikr/4πr

function call{k,n}(::FirstIntegral{Helmholtz{k},n}, X::Number)
```

[1] The `exprel` function used here is available in the external GSL package; see appendix B.

```
    ikX = im*k*X
    return exp(ikX) * exprel(n, -ikX) / (4π*n*X)
end

# magnetic field integral equation
type MFIE{k} <: APowerLaw{-3,-3}; end  # (ikr − 1) ∗ e^{ikr}/4πr^3

function call{k,n}(::FirstIntegral{MFIE{k},n}, X::Number)
    ikX = im*k*X
    return exp(ikX) * (im*k*exprel(n-1,-ikX)/((n-1)*X) -
                       exprel(n-2,-ikX)/((n-2)*X^2)) / (4π*X)
end
```

It is possible to implement `FirstIntegral` for the general case of any `APowerLaw` using numerical integration, however this procedure is too slow to be useful. Instead, we would like to compute the integral for a range of parameter values in advance, construct a Chebyshev approximation, and use efficient polynomial evaluation at run time. Julia's staged methods provide an easy way to automate this process. Appendix B presents a full working implementation of the general case, computing Chebyshev coefficients at compile time. The final generated code is a straight-line sequence of adds and multiplies. Here we show a rough outline of how the method works:

```
@generated function call{P<:APowerLaw,n}(K::FirstIntegral{P,n},
                                          X::Real)
    # compute Chebyshev coefficients c based on K
    # ...
    quote
        X <= 0 && throw(DomainError())
        ξ = 1 - (X + $(2^(1/q)))^$q
        C = @evalcheb ξ $(c...)
        return $p < 0 ? C * (X^$p + $(s^p)) : C
    end
end
```

The method signature reflects the shape of the problem: the user might implement any kernel at all, but if it does not take the form of an `APowerLaw{p}` then this numerical procedure is not useful. `p` must be known, and cannot be easily extracted from an arbitrary opaque function. If the interface allowed an opaque function to be passed, the value of `p` would need to be passed separately, which de-couples the property from the function, and

requires a custom lookup table for caching generated code.

One might hope that with a sufficient amount of constant propagation and functional purity analysis, a compiler could automatically move computation of the Chebyshev coefficients to compile time. That does indeed seem possible. The point of `@generated`, then, is to make it so easy to get the desired staging that one does not have to worry about what amazing feats the compiler might be capable of. Without it, one would face a difficult debugging challenge when the compiler, for whatever opaque reasons, did not perform the desired optimizations.

To complete the picture, here is pseudocode for how this library would be used to solve a boundary element problem over a triangle mesh. The three cases inside the loop are a translation of the three formulas from figure 1 in [97] (only the first formula's code is shown in full for simplicity):

```
FP = [(FirstIntegral{K,i}(), polynomial(i)) for i in N]
for trianglepair in mesh
    i, j = indexes based on triangles
    if common(trianglepair)
        M[i,j] = integrate(y->sum([F(y)*P(y) for (F,P) in FP]),
                           0, 1)
    elseif commonedge(trianglepair)
        # a 2d integral
    elseif commonvertex(trianglepair)
        # a 3d integral
    else
        # separate triangles
    end
end
```

5.8 Beating the incumbents

There is anecdotal evidence that libraries written in Julia are occasionally faster than their equivalents in established languages like C++, or than built-in functions in environments like MATLAB [82]. Here we collect a few examples.

Special functions (e.g. Bessel functions, the error function, etc.) are typically evaluated

using polynomial approximations. Traditional libraries often read polynomial coefficients from arrays. When we wrote some of these functions in Julia we used a macro to expand the polynomials in line, storing the coefficients in the instruction stream, leading to better performance. There is an efficient algorithm for evaluating polynomials at complex-valued arguments [67], but it is fairly subtle and so not always used. If the code can be generated by a macro, though, it is much easier to take advantage of, and Julia's `@evalpoly` does so.

Julia's function for generating normally distributed random numbers is significantly faster than that found in many numerical environments. We implemented the popular ziggurat method [78], which consists of a commonly used fast path and a rarely used more complicated procedure. The fast path and check for the rare case can be put in a small function, which can be inlined at call sites in user code. We attribute most of the performance gain to this trick. This demonstrates that removing "glue code" overhead can be worthwhile.

Julia has long used the Grisu algorithm for printing floating-point numbers [75], at first calling the original implementation in C++. An opportunity to do an unusually direct comparison arose when this code was ported to Julia [94]. The Julia implementation is at worst 20% slower, but wins on productivity: the C++ code is around 6000 lines, the Julia code around 1000 lines.

Chapter 6

Conclusion

We began by observing that technical computing users have priorities that have rarely been addressed directly by programming languages. Arguably, even many of the current technical computing languages only address these priorities for a subset of cases. In some sense all that was missing was an appropriate notion of partial information.

partial information implies - describing sets of values, therefore code applicability - a performance model, capturing what sorts of information the compiler can determine - staging. the whole idea of generating code depends on knowing something about what needs to be done, but not everything, so that it can be reused enough to be worthwhile. therefore partial information.

of course this sounds a lot like a type system, but the point is that this idea can perform important roles at run time.

6.1 Performance

If we are interested in abstraction, then type specialization is the first priority for getting performance. However in a broader view there is much more to performance. For example "code selection" appears in a different guise in the idea of *algorithmic choice* [6, 7]: at any point in a program, we might have a choice of several algorithms and want to automatically

pick the fastest one. Julia's emphasis on writing functions in small pieces and describing applicability seems naturally suited to this kind of approach.

6.2 Implementation issues

As with any practical system, we must begin the process of trying to get back what our design initially trades away.

For example, the specialization-based polymorphism we use does not support separate compilation. Fortunately that's never stopped anyone before: C++ templates have the same problem, but separately compiled modules can still be generated. We plan to do the same, with the same cost of re-analyzing all relevant code for each module. Another approach is to use layering instead of fully separate modules. Starting with the core language, one next identifies and compiles a useful base library. That layer is then considered "sealed" and can no longer be extended — it effectively becomes part of the language. Now applications using this library can be compiled more efficiently. More layers can be added (slowest-changing first) to support more specific use cases. This is essentially the "telescoping languages" approach [66].

6.3 Future work

- formalize more details of dispatch, meet, join, widen
- product domains, how to incorporate finer types more smoothly
- how to incorporate static checks? [55]
- separate compilation
- function types
- shared memory, cilk

- can we implement BLAS?
- incorporate autotuning?

6.4 Project status

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(),(),false); AnyT.name.super = AnyT
type BottomTy <: Ty; end; const BottomT = BottomTy()
const TupleName = TypeName(); TupleName.super = AnyT

super(t::TagT) =
  t.name===TupleName ? AnyT : inst(t.name.super, t.params...)

## type application
inst(t::TagT) = t
inst(t::UnionAllT, param) = subst(t.T, Dict(t.var => param))
inst(t::UnionAllT, param, rest...) = inst(inst(t,param), rest...)

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, merge(env, Dict(t.var=>newVar))))
end

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

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

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

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

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

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

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

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

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

    found = issub(x, y, env)

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

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

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

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

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

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

issub(a::Ty, b::Var, env) = var_gt(b, a, env)
issub(a::Var, b::Ty, env) = var_lt(a, b, env)
function issub(a::Var, b::Var, env)
  a === b && return true
  aa = env.vars[a]; bb = env.vars[b]
  if aa.right
    bb.right && return issub(bb.ub, bb.lb, env)
```

```
    return var_lt(a, b, env)
  else
    if !bb.right   # check ∀a,b . a<:b
      return issub(aa.ub, b, env) || issub(a, bb.lb, env)
    end
    return var_gt(b, a, env)
  end
end

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

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

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

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

Appendix B

Staged numerical integration

```
abstract AbstractKernel

# any kernel X^p for X ≪ s and X^q for X ≫ s
abstract APowerLaw{p,q,s} <: AbstractKernel

type FirstIntegral{K<:AbstractKernel,n}; end

type PowerLaw{p} <: APowerLaw{p,p}; end # r^p power law

# N chebyshev points (order N) on the interval (-1,1)
chebx(N) = [cos(π*(n+0.5)/N) for n in 0:N-1]

import GSL
exprel(n, x) = GSL.sf_exprel_n(n, x)

# N chebyshev coefficients for vector of f(x) values on chebx
# points x
function chebcoef(f::AbstractVector)
    a = FFTW.r2r(f, FFTW.REDFT10) / length(f)
    a[1] /= 2
    return a
end

# given a function f and a tolerance, return enough Chebyshev
# coefficients to reconstruct f to that tolerance on (-1,1)
function chebcoef(f, tol=1e-13)
    N = 10
    local c
```

```
    while true
        x = chebx(N)
        c = chebcoef(float([f(y) for y in x]))
        # look at last 3 coefs, since individual c's might be 0
        if max(abs(c[end]),
               abs(c[end-1]),
               abs(c[end-2])) < tol * maxabs(c)
            break
        end
        N *= 2
    end
    return c[1:findlast(v -> abs(v)>tol, c)] # shrink to min length
end

# given cheb coefficients a, evaluate them for x in (-1,1) by
# Clenshaw recurrence
function evalcheb(x, a)
    -1 ≤ x ≤ 1 || throw(DomainError())
    b_{k+1} = b_{k+2} = zero(x)
    for k = length(a):-1:2
        b_k = a[k] + 2x*b_{k+1} - b_{k+2}
        b_{k+2} = b_{k+1}
        b_{k+1} = b_k
    end
    return a[1] + x*b_{k+1} - b_{k+2}
end

# inlined version of evalcheb given coefficents a, and x in (-1,1)
macro evalcheb(x, a...)
    # Clenshaw recurrence, evaluated symbolically:
    b_{k+1} = b_{k+2} = 0
    for k = length(a):-1:2
        b_k = esc(a[k])
        if b_{k+1} != 0
            b_k = :(muladd(t2, $b_{k+1}, $b_k))
        end
        if b_{k+2} != 0
            b_k = :($b_k - $b_{k+2})
        end
        b_{k+2} = b_{k+1}
        b_{k+1} = b_k
    end
    ex = esc(a[1])
    if b_{k+1} != 0
        ex = :(muladd(t, $b_{k+1}, $ex))
```

```
    end
    if b_{k+2} != 0
        ex = :($ex - $b_{k+2})
    end
    Expr(:block, :(t = $(esc(x))), :(t2 = 2t), ex)
end

# extract parameters from an APowerLaw
APowerLaw_params{p,q,s}(::APowerLaw{p,q,s}) = (p, q, s)

@generated function call{P<:APowerLaw,n}(::FirstIntegral{P,n},
                                         X::Real)
    # compute the Chebyshev coefficients of the rescaled 𝒦_n
    K = P()
    p, q, s = APowerLaw_params(K)
    𝒦_n = X->quadgk(w -> w^n * K(w*X), 0, 1, abstol=1e-14,
                    reltol=1e-12)[1]
    # scale out X ≪ s singularity
    L_n = p < 0 ? X -> 𝒦_n(X) / (s^p + X^p) : 𝒦_n
    q > 0 && throw(DomainError()) # can't deal with growing kernels
    qinv = 1/q
    c = chebcoef(ξ -> L_n((1-ξ)^qinv - 2^qinv), 1e-9)
    # return an expression to evaluate 𝒦_n via C(ξ)
    quote
        X <= 0 && throw(DomainError())
        ξ = 1 - (X + $(2^qinv))^$q
        C = @evalcheb ξ $(c...)
        return $p < 0 ? C * (X^$p + $(s^p)) : C
    end
end
```

Bibliography

[1] *Getting Started with MATLAB. Version 5.* MATHWORKS Incorporated, 1998.

[2] Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst. 13*, 2 (Apr. 1991), 237–268.

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

[4] 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.

[5] An, J.-h. D., Chaudhuri, A., Foster, J. S., and Hicks, M. Dynamic inference of static types for ruby. *SIGPLAN Not. 46* (Jan. 2011), 459–472.

[6] Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 38–49.

[7] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2014), PACT '14, ACM, pp. 303–316.

[8] Baker, H. G. The nimble type inferencer for Common Lisp-84. Tech. rep., Tech. Rept., Nimble Comp, 1990.

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

[10] Baumgartner, G., Auer, A., Bernholdt, D. E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R. J., Hirata, S., Krishnamoorthy, S., et al. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE 93*, 2 (2005), 276–292.

[11] Beer, R. D. Preliminary report on a practical type inference system for Common Lisp. *SIGPLAN Lisp Pointers 1* (June 1987), 5–11.

[12] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. Cython: The best of both worlds. *Computing in Science and Engg. 13*, 2 (Mar. 2011), 31–39.

[13] Benzaken, V., Castagna, G., and Frisch, A. CDuce: an XML-centric general-purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming* (Uppsala, Sweden, 2003), ACM Press, pp. 51–63.

[14] Bezanson, J., Chen, J., Karpinski, S., Shah, V., and Edelman, A. Array operators using multiple dispatch. In *Proc. ACM SIGPLAN Int. Work. Libr. Lang. Compil. Array Program. - ARRAY'14* (New York, New York, USA, 2014), ACM, pp. 56–61.

[15] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common Lisp Object System specification. *SIGPLAN Not. 23* (September 1988), 1–142.

[16] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (New York, NY, USA, 2009), ICOOOLPS '09, ACM, pp. 18–25.

[17] 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.

[18] Bonnet, M. *Boundary Integral Equation Methods for Solids and Fluids*. Wiley, Chichester, England, 1999.

[19] Boyd, J. P. *Chebyshev and Fourier Spectral Methods*, 2nd ed. Dover, New York, NY, USA, 2001.

[20] Cameron, N., and Drossopoulou, S. On subtyping, wildcards, and existential types. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs* (New York, NY, USA, 2009), FTfJP '09, ACM, pp. 4:1–4:7.

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

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

[23] Castagna, G., and Frisch, A. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (New York, NY, USA, 2005), PPDP '05, ACM, pp. 198–199.

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

[25] 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.

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

[27] Chamberlain, B. L., Callahan, D., and Zima, H. P. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[28] Chambers, C. Object-oriented multi-methods in cecil. In *Proceedings of the European Conference on Object-Oriented Programming* (London, UK, 1992), Springer-Verlag, pp. 33–56.

[29] Chambers, C. The cecil language specification and rationale: Version 2.1.

[30] Chambers, C. The diesel language specification and rationale: Version 0.2.

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

[32] Chew, W. C., Jin, J.-M., Michielssen, E., and Song, J., Eds. *Fast and Efficient Algorithms in Computational Electromagnetics*. Artech, Norwood, MA, 2001.

[33] Chew, W. C., Tong, M. S., and Hu, B. *Integral Equation Methods for Electromagnetic and Elastic Waves*. Synthesis Lectures on Computational Electromagnetics. Morgan and Claypool, San Rafael, CA, 2007.

[34] Cline, M. P., and Lomow, G. A. *C++ FAQs*. Addison-Wesley, 1995.

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

[36] 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.

[37] DeMichiel, L., and Gabriel, R. The Common Lisp Object System: An overview. In *ECOOPâĂŹ 87 European Conference on Object-Oriented Programming*, J. BÃlzivin, J.-M. Hullot, P. Cointe, and H. Lieberman, Eds., vol. 276 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1987, pp. 151–170.

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

[39] DeVito, Z., Ritchie, D., Fisher, M., Aiken, A., and Hanrahan, P. First-class runtime generation of high-performance types using exotypes. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 77–88.

[40] Dragos, I., and Odersky, M. Compiling generics through user-directed type specialization. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (New York, NY, USA, 2009), ICOOOLPS '09, ACM, pp. 42–47.

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

[42] Ernst, M. D., Kaplan, C. S., and Chambers, C. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming* (Brussels, Belgium, July 20-24, 1998), pp. 186–211.

[43] Falkoff, A. D., and Iverson, K. E. The design of APL. *SIGAPL APL Quote Quad 6* (April 1975), 5–14.

[44] Fischer, K. SI units package, 2014. `https://github.com/Keno/SIUnits.jl`.

[45] Fourer, R., Gay, D. M., and Kernighan, B. *AMPL*, vol. 119. Boyd & Fraser, 1993.

[46] Frigo, M., and Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[47] Frisch, A., Castagna, G., and Benzaken, V. Semantic subtyping. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on* (2002), pp. 137–146.

[48] 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.

[49] Gapeyev, V., and Pierce, B. Regular object types. In *ECOOP 2003 âĂŞ Object-Oriented Programming*, L. Cardelli, Ed., vol. 2743 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 151–175.

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

[51] Garcia, R., and Lumsdaine, A. MultiArray: a C++ library for generic programming with arrays. *Software: Practice and Experience 35*, 2 (2005), 159–188.

[52] Garg, R., and Hendren, L. Just-in-time shape inference for array-based languages. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (New York, NY, USA, 2014), ARRAY'14, ACM, pp. 50:50–50:55.

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

[54] Grcevski, N., Kielstra, A., Stoodley, K., Stoodley, M. G., and Sundaresan, V. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Virtual Machine Research and Technology Symposium* (2004), pp. 151–162.

[55] Hanson, L. TypeCheck.jl, 2014. `https://github.com/astrieanna/TypeCheck.jl`.

[56] Holy, T. Traits using multiple dispatch, 2014. `https://github.com/JuliaLang/julia/issues/2345#issuecomment-54537633`.

[57] Hosoya, H., and Pierce, B. C. XDuce: A typed XML processing language. In *Third International Workshop on the Web and Databases (WebDB2000)* (2000), vol. 1997, Springer, pp. 226–244.

[58] Hosoya, H., Vouillon, J., and Pierce, B. C. Regular expression types for xml. In *ACM SIGPLAN Notices* (2000), vol. 35, ACM, pp. 11–22.

[59] 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.

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

[61] Joisha, P. G., and Banerjee, P. An algebraic array shape inference system for matlab®. *ACM Trans. Program. Lang. Syst. 28*, 5 (Sept. 2006), 848–907.

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

[63] Karpinski, S., Holy, T., and Danisch, S. Parametric color types, 2014. `http://github.com/JuliaLang/Color.jl/issues/42`.

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

[65] Keller, G., Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S., and Lippmeier, B. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2010), ICFP '10, ACM, pp. 261–272.

[66] Kennedy, K., Broom, B., Cooper, K., Dongarra, J., Fowler, R., Gannon, D., Johnsson, L., Mellor-Crummey, J., and Torczon, L. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing 61*, 12 (2001), 1803 – 1826.

[67] Knuth, D. E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. Addison-Wesley, 1969.

[68] 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).

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

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

[71] Lin, C., and Snyder, L. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., vol. 768 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1994, pp. 96–114.

[72] Liu, Y. *Fast Multipole Boundary Element Method: Theory and Applications in Engineering*. Cambridge University Press, New York, NY, USA, 2014.

[73] Logg, A., Mardal, K.-A., and Wells, G. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Lecture Notes in Computational Science and Engineering. Springer, New York, NY, USA, 2012.

[74] Logg, A., Ølgaard, K. B., Rognes, M. E., and Wells, G. N. *FFC: the FEniCS Form Compiler*. Springer, 2012, ch. 11.

[75] Loitsch, F. Printing floating-point numbers quickly and accurately with integers. *SIGPLAN Not. 45*, 6 (June 2010), 233–243.

[76] Lubin, M., and Dunning, I. Computing in operations research using julia. *INFORMS Journal on Computing 27*, 2 (2015), 238–248.

[77] Ma, K.-L., and Kessler, R. R. TICLâĂŤa type inference system for Common Lisp. *Software: Practice and Experience 20*, 6 (1990), 593–623.

[78] Marsaglia, G., and Tsang, W. W. The ziggurat method for generating random variables. *Journal of Statistical Software 5*, 8 (10 2000), 1–7.

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

[80] MathWorks, Inc. *MATLAB Manual: linspace*, r2014b ed. Natick, MA.

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

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

[83] Merrill, J. Julia vs Mathematica (Wolfram language): high-level features, 2014. `https://groups.google.com/d/topic/julia-users/EQH5LJZqe8k/discussion`.

[84] Miller, E. Why I'm betting on julia, 2014. `http://www.evanmiller.org/why-im-betting-on-julia.html`.

[85] Millstein, T., Frost, C., Ryder, J., and Warth, A. Expressive and modular predicate dispatch for Java. *ACM Trans. Program. Lang. Syst. 31*, 2 (Feb. 2009), 7:1–7:54.

[86] Mitchell, S., OâĂŹSullivan, M., and Dunning, I. PuLP: a linear programming toolkit for python. *The University of Auckland, Auckland, New Zealand* (2011).

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

[88] Morandat, F., Hill, B., Osvald, L., and Vitek, J. Evaluating the design of the r language. In *ECOOP 2012 - Object-Oriented Programming*, J. Noble, Ed., vol. 7313 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 104–131.

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

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

[91] Muschevici, R., Potanin, A., Tempero, E., and Noble, J. Multiple dispatch in practice. *SIGPLAN Not. 43* (October 2008), 563–582.

[92] Ofenbeck, G., Rompf, T., Stojanov, A., Odersky, M., and Püschel, M. Spiral in scala: Towards the systematic construction of generators for performance libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences* (New York, NY, USA, 2013), GPCE '13, ACM, pp. 125–134.

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

[94] Quinn, J. Grisu algorithm in julia, 2014. `https://github.com/JuliaLang/julia/pull/7291`.

[95] Rathgeber, F., Ham, D. A., Mitchell, L., Lange, M., Luporini, F., McRae, A. T., Bercea, G.-T., Markall, G. R., and Kelly, P. H. Firedrake: automating the finite element method by composing abstractions. *Submitted to ACM TOMS* (2015).

[96] Rathgeber, F., Markall, G., Mitchell, L., Loriant, N., Ham, D., Bertolli, C., and Kelly, P. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* (Nov 2012), pp. 1116–1123.

[97] Reid, M. T. H., White, J. K., and Johnson, S. G. Generalized Taylor–Duffy method for efficient evaluation of Galerkin integrals in boundary-element method computations. *IEEE Transactions on Antennas and Propagation PP* (November 2014), 1–16.

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

[99] Rompf, T., and Odersky, M. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (New York, NY, USA, 2010), GPCE '10, ACM, pp. 127–136.

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

[101] Scott, D. Data types as lattices. *Siam Journal on computing 5*, 3 (1976), 522–587.

[102] Scott, D., and Strachey, C. *Toward a mathematical semantics for computer languages*, vol. 1. Oxford University Computing Laboratory, Programming Research Group, 1971.

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

[104] 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.

[105] Snir, M. *MPI–the Complete Reference: The MPI core*, vol. 1. MIT press, 1998.

[106] Steele, G. L. *Common LISP: the language*. Digital press, 1990, ch. 4.

[107] Stone, C. A., and Harper, R. Deciding type equivalence in a language with singleton kinds. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2000), POPL '00, ACM, pp. 214–227.

[108] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., and Nakatani, T. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst. 27*, 4 (July 2005), 732–785.

[109] Sussman, G. J., and Wisdom, J. *Structure and Interpretation of Classical Mechanics*. MIT Press, Cambridge, MA, USA, 2001.

[110] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures* (2011), ACM, pp. 117–128.

[111] Tate, R., Leung, A., and Lerner, S. Taming wildcards in java's type system. *SIGPLAN Not. 46*, 6 (June 2011), 614–627.

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

[113] Vasilev, V., Canal, P., Naumann, A., and Russo, P. Cling–the new interactive interpreter for ROOT 6. In *Journal of Physics: Conference Series* (2012), vol. 396, IOP Publishing, p. 052071.

[114] Wadler, P. The expression problem. *Java-genericity mailing list* (1998).

[115] Wehr, S., and Thiemann, P. Subtyping existential types. *10th FTfJP, informal proceedings* (2008).

[116] Wilbers, I. M., Langtangen, H. P., and Ødegård, Å. Using cython to speed up numerical python programs. *Proceedings of MekIT 9* (2009), 495–512.

[117] Zienkiewicz, O. C., Taylor, R. L., and Zhu, J. Z. *The Finite Element Method: Its Basis and Fundamentals*, 7th ed. Butterworth-Heinemann, Oxford, England, 1967.