

Abstraction in Technical Computing

by

Jeffrey Werner Bezanson

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
May 1, 2015

Certified by
Alan Edelman
Professor
Thesis Supervisor

Accepted by
Leslie Kolodziejcki
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 1, 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	7
List of Tables	8
1 Introduction	9
1.1 The technical computing problem	11
1.2 Solution space	12
1.2.1 The excess power problem	13
1.2.2 The divergence problem	14
1.2.3 Staged programming	15
1.3 Contributions of this thesis	15
2 The nature of technical computing	17
2.1 What is a technical computing environment?	17
2.1.1 Case study: Vandermonde matrices	19
2.2 Why dynamic typing?	22
2.2.1 Mismatches with mathematical abstractions	22
2.2.2 I/O and data-oriented programming	23
2.3 Tradeoffs in current designs	23
2.3.1 Vectorization	24
2.3.2 Data representation	26
2.4 A compiler for every problem	27

2.5	Social phenomena	28
3	Available technology	29
3.1	The language design space	29
3.1.1	Classes vs. functions	30
3.1.2	Separate compilation vs. performance	30
3.1.3	Parametric vs. ad-hoc polymorphism	30
3.1.4	Static checking vs. flexibility	31
3.1.5	Modularity vs. large vocabularies	31
3.1.6	Eliminating vs. embracing tagged data	32
3.1.7	Data hiding vs. explicit memory layout	32
3.1.8	Dynamic dispatch is key	32
3.2	Dispatch systems	37
3.2.1	Example: lattices	37
3.2.2	Symbolic programming	38
3.2.3	Predicate dispatch	40
3.3	Domain theory	41
3.3.1	Monotonicity	42
3.4	Subtyping	43
4	The Julia approach	44
4.1	Core calculus and data model	44
4.2	Type system	47
4.2.1	Examples	51
4.2.2	Type constructors	51
4.2.3	Subtyping	52
4.2.4	Type system variants	55
4.3	Dispatch mechanism	55
4.3.1	Type and method caches	56
4.3.2	Specificity	56

4.3.3	Ambiguities	57
4.3.4	Constructors	57
4.4	Generic programming	57
4.5	Staged programming	58
4.6	Higher-order programming	59
4.6.1	Problems for code selection	60
4.6.2	Problems for code specialization	61
4.6.3	Possible solutions	63
4.6.4	Implementing map	64
4.7	Performance model	66
4.7.1	Type inference	66
4.7.2	Specialization	66
5	Case studies	69
5.1	Conversion and comparison	69
5.2	Numeric types and embeddings	71
5.3	Multidimensional array indexing	76
5.4	Array views	79
5.5	Numerical linear algebra	79
5.6	Units	81
5.7	Boundary-element method	83
5.8	Algebraic modeling with JuMP	88
5.9	Beating the incumbents	90
6	Conclusion	91
A	Subtyping algorithm	93
	Bibliography	98

List of Figures

1-1	49 technical computing languages	10
2-1	Performance of vectorized code	25
3-1	A small Julia library for lattices	39
4-1	Subtyping algorithm	52
4-2	An implementation of map	65

List of Tables

1.1	Programming language priorities	11
2.1	Performance of vander	22
3.1	A coarse classification of languages	29
4.1	Degree of reuse of function types	62
4.2	Multiple dispatch use statistics	67
5.1	Performance of linear programming tools	90

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 is where you go when you need this level of flexibility. Language design is associated with the highest level of abstraction. 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

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

computing domain uses general purpose languages. Effective scientific libraries extensively employ polymorphism, custom operators, and compile time abstraction. Code generation approaches (writing a program that writes the needed program) are unusually common. Most of these techniques are presently fairly difficult to use, and so programmers working at this level give up the notational conveniences of the purpose-built languages above.

This thesis addresses the essential challenges created by this state of affairs. We argue that the situation can be improved considerably by abstracting, with a new language.

Compiler techniques, library design, high-performance computational kernels, new algorithms, and approaches to parallelism are all important to technical computing. However these sorts of technologies can usually be applied to multiple languages, as has happened in the C and Fortran language families. However in conjunction with a new language, these technologies can flourish. Indeed, we believe that language design has reached a level of importance that even (perhaps temporarily) supersedes the traditional focus of cache utilization and the use of matrix operations.

In brief, our idea 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. This notion subsumes method dispatch, function overloading, and

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

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

potentially even branches. When such mechanisms are used in technical computing, there is nearly always a corresponding need to *specialize* code for specific cases to obtain performance. Polymorphic languages sometimes support forms of specialization, but often only through a designated mechanism (e.g. templates), or deployed rather 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 dynamic type-based dispatch mechanism. The specific design is identified via subtyping theories and closure under data-flow operations.

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 [69], with a high-level productivity language layered on top of libraries written in lower-level languages. Seeking a similar tradeoff, others have gone as far as writing a C++ interpreter [70].

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 [28] 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 [14], for instance), but it is still too early to call a winner between these two approaches (if, indeed, there even needs to be a winner).

Here we will use a dynamically typed approach, with approximate type inference (for an excellent discussion of the many meanings of the word “type” see [37]). There are several reasons for this. First, insufficient static checking is most likely not the current limiting factor in scientific computing productivity. Second, some idioms in this domain appear to be inherently, or at least naturally, dynamically typed (as we will explore in chapter 2). Third, there has been a sense that “dynamic” or “scripting” language users do not want to hear about types: they are thought of as a distraction and associated with verbosity and

nuisance compiler errors. We hope to contribute an example of a language where types are useful and not burdensome.

1.2.1 The excess power problem

We claim that the amount of information that can be statically inferred exceeds most dynamic languages' capacity to exploit it. Much work has been done on program analysis and optimization techniques for dynamically typed languages. When static analyses (often incorporating run time information) are applied to dynamically typed programs, it is typically possible to recover a significant amount of type information (TODO cite). What, then, can one do with this information? If the goal is performance, various partial evaluations can be done: generating code without type checks, removing branches, type-specializing the storage of variables, and compile time method lookup are all valuable and yield large real-world gains.

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

This “excess power” problem applies to data structures as well. For example, static or run time analysis might reveal that a certain array can be represented as a native Int32 array [8]. If this information is not reflected in the source language, then certain uses like passing data to native code become unnecessarily more complicated. And if one is going to implement homogeneous arrays anyway, why not let programmers request them?

Some levels of performance are difficult to reach with implicitly specialized code and data. Given the knowledge that an array contains only Int32 data, we might want to go beyond essential optimizations like storing intermediate values in registers, and actually use different

algorithms. For example, in Miller-Rabin primality testing, checking three “witness” values suffices for all 32-bit arguments, but up to seven values might be needed for 64-bit arguments .(TODO cite) In cryptographic applications, exploiting this difference in an inner loop could bring significant benefits.

cite

1.2.2 The divergence problem

Another problem that occurs when analyzing programs with complex type behavior is divergence: the analysis is likely to infer an overly-large result from failing to eliminate enough possible behaviors. Narrowing the inferred types requires some extra source of type information. Multi-method signatures work well for this purpose.

past approaches include: optional user annotations, or a built-in database of function type behavior, determined empirically or analytically. to solve the problem in general, library behavior should not be built-in to the compiler. however, when functions that are both highly complex and performance critical are implemented at the library level it can be difficult to ensure they are sufficiently optimized.

However the divergence problem also places a limit on what kinds of dispatch specifications are useful to program analyses: some sets of values tend to be more robust under execution than others.

Would love an example if possible

Why do we prefer method signatures to imperative logic?

- it's extensible - its declarative style is more compact - allows writing functions in small pieces the more descriptive the types, the smaller the pieces might make compiler more efficient; removes large bits of code from consideration just from looking at types. don't need to analyze a large function just to discover that only a small part of it remains. - clearer performance model. it's hard for programmer to guess when the compiler will statically evaluate an arbitrary expression - allows defining semantics about things that are optimized, but **not** compile-time constants. method table can be (ab)used as a mutable lookup table structure.

1.2.3 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 semantically dynamically typed, has a canonical (and not optional!) type analysis stage. This is a natural place to hook in code generators: semantic (not just syntactic) information is available, but machine code has not been generated yet. In general, this semantic information can also include run time (and always, at least, approximate run time) information. Since the types used by dispatch have rich, 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. Our approach is closely related to exotypes [20].

1.3 Contributions of this thesis

Our first contribution is a discussion of the nature of technical computing that suggests which language-level abstractions might best support real use cases. Simply put technical

computing has by and large happened without sufficient probing and analysis as to what it is. Our looking at the space in this manner has lead to a novel design. The success of this design is resonating with users worldwide.

Based on the theory that technical computing is characterized by complex operators and particular combinations of binding time behavior.

To our knowledge, the first technical computing system written in a single language. Past systems have applied specialization to a set of user-level routines. We show that it can be used for an entire system.

Second, we introduce the idea of integrated code selection and specialization.

Using data flow types to drive the design leads to novel dispatch features: - diagonal dispatch - unifying variadic methods with regular types - predictable performance without giving up flexibility

Easy polymorphism

- library writers can decide what to put at the type level without affecting users - a single mechanism to cover overloading, dispatch, and specialization - specialize by default (makes static polymorphism less of a black art)

Subtyping algorithm

Library design issues

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.” [52] An additional contribution of this thesis is the application of this approach to features of technical computing environments that have not been subject to such analysis before.

Chapter 2

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 [48], R [35], Mathematica [45], Octave [53], Python (with NumPy) [69], and SciLab [32] 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 [47]. 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 [69] 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(MulFun(), 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 significantly faster in this case; it may be due to NumPy’s more general algorithm that can accumulate along any dimension. In that sense the comparison is not totally fair. 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.

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.

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 5.5.

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` [66], which is 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 lots of 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 Tradeoffs in current designs

Current systems accept certain design tradeoffs 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 much 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 [51]. 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

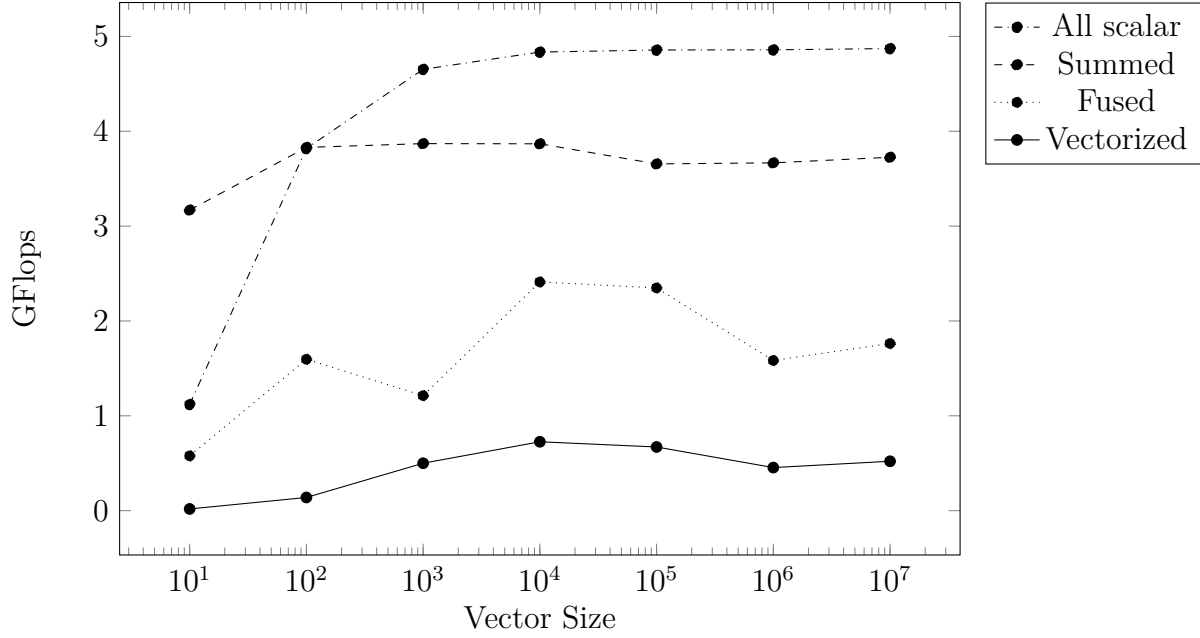


Figure 2-1: Performance of evaluating $x^2 + x - 1$ over a double precision vector of length n . Operation repeated $10^7/n$ times. (Core i7-3517U 1.9GHz, 4MB cache, 8GB RAM)

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 [23] and ZPL [42] 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 [46]). 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 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 [49]. Jason Merrill discovered a Mathematica [45] 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 entirely true. 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* [8], 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 [6] 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 [56] and FEniCS [43] for FEM problems, PyOP2 [57] for general computations on meshes, FFTW [26] for signal processing, and Pochoir [68] 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 also 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 Julia’s “easy polymorphism” design naturally blends these two 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.” [50]

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 either object can be used in certain contexts. We will discuss more extensive examples of such situations in chapter 5.

Chapter 3

Available technology

3.1 The language design space

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

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

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

Table 3.1:

result the top-left corner of this design space has been under-explored.

3.1.1 Classes vs. functions

Technical computing is function oriented.

in C++ to “extend” an existing type you have to use another mechanism, overloading. however then you also need to accept different semantics, static resolution.

3.1.2 Separate compilation vs. performance

Writing the signature of a generic method that needs to be separately compiled, as in Java for example, can be a difficult exercise. The programmer must use the type system to write down sufficient conditions on all arguments. (TODO example from [29]) If, however, we are going to specialize the method, the compiler can analyze it using actual types from call sites, and see for itself whether the method works in each case (this is how C++ templates are type-checked; they are analyzed again for each instantiation).

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

3.1.3 Parametric vs. ad-hoc polymorphism

Can we have a definition of ad-hoc polymorphism and talk about its relation to multiple dispatch? and point to a technical computing example?

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

Next we can imagine that the compiler might not optimally specialize certain definitions, and that the programmer would like to provide hints or hand-written implementations to speed up special cases. For example, imagine a function that traverses a generic array. A

compiler-generated specialization might inline a specific array types's indexing operations, but a human might further realize that the loop order should be switched for certain arrays types based on their storage order.

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

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

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.

3.1.4 Static checking vs. flexibility

3.1.5 Modularity vs. large vocabularies

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

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

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

3.1.6 Eliminating vs. embracing tagged data

3.1.7 Data hiding vs. explicit memory layout

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

The loss of encapsulation due to multi-methods weighed in [9] is less of a problem for technical computing, and in some cases even advantageous.

3.1.8 Dynamic dispatch is key

Comment: Really nice!

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

In class-based OO there is essentially *no way* to create an operation that dispatches on existing types (the expression problem [71]). This clearly does not match technical comput-

ing, where most programs deal with the same few types (e.g. number, array), and might sensibly want to write new operations that dispatch on them.

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

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

```
while running
  for a in animals
    b = nearby_animal(a)
    if a isa Rabbit
      if b isa Wolf then run(a)
      if b isa Rabbit then mate(a,b)
    else if a isa Wolf
      if b isa Rabbit then eat(a,b)
      if b isa Wolf then follow(a,b)
    end
  end
end
end
```

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

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

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

```

        end
    end

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

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

```

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

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

has <: been defined yet?

```

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

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

```

```

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

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

```

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

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

```

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

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

```

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

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

in a special way based on the first argument? It seems arbitrary, and indeed we can remove the special treatment:

```
abstract class Animal
end

class Rabbit <: Animal
end

class Wolf <: Animal
end

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

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

Here we made two major changes: the methods have been moved “outside” of any classes, and all arguments are listed explicitly. This change has fairly significant implications. Since methods no longer need to be “inside” classes, there is no syntactic limit to where definitions may appear. Now it is easier to add new methods after a class has been defined. Methods also now naturally operate on combinations of objects, not single objects.

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

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

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

We can also do diagonal dispatch:

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

3.2 Dispatch systems

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

For this reason, C++ programs are not fully generic: it is difficult to substitute new behaviors for every part of a program. Some languages take generic programming much further. For example, in MATLAB, *every* function is effectively generic, and can be overloaded by new classes. This enables a programmer to “intercept” every function call (and therefore, essentially everything a program does)

3.2.1 Example: lattices

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

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

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

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

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

Join of two incomparables in general lattices does not have to be top

sentence
grammar?

3.2.2 Symbolic programming

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

Love it!

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

what does `===` mean here?

```
abstract LatticeElement

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

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

const  $\top$  = TopElement()
const  $\perp$  = BotElement()

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

 $\sqcup$ (x::LatticeElement, y::LatticeElement) = # join
    (x <= y ? y : y <= x ? x :  $\top$ )

 $\sqcap$ (x::LatticeElement, y::LatticeElement) = # meet
    (x <= y ? x : y <= x ? y :  $\perp$ )
```

Figure 3-1: A small Julia library for lattices

values and structures. These systems are typically powerful enough to concisely define the kinds of behaviors we are interested in.

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

Wish you
would say
more!

3.2.3 Predicate dispatch

Predicate dispatch is a powerful object-oriented mechanism that allows methods to be selected based on arbitrary predicates [22]. 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 [67], have actually included predicates as types. However, such a type system is obviously undecidable, since it requires computing the predicates themselves or, even worse, computing predicate implication.²

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

In considering the problems of predicate dispatch for code specialization, we seem to

²Many type systems involving bounded quantification, such as system $F_{<}$, are already undecidable [55]. However, they seem to terminate for most practical programs, and also admit minor variations that yield decidable systems [15]. It is fair to say they are “just barely” undecidable, while predicates are “very” undecidable.

be up against a fundamental obstacle: some sets of values are simply more robust under evaluation than others. Programs that map integers to integers abound, but programs that map, say, even integers to even integers are rare to the point of irrelevance.

3.3 Domain theory

In the late 1960s Dana Scott and Christopher Strachey asked how to assign meanings to programs, which otherwise just appear to be lists of symbols [63]. 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” in this theory is a partial order of sets of values that a program might manipulate. Domain theory models computation as follows: a program starts with no information, the lowest point in the partial order (“bottom”). Computation steps accumulate information, gradually moving higher through the order. The advantage of this model, in essence, is that it provides a way to think about the meaning of a program without running the program. Even the “result” of a non-terminating program has a representation — the bottom element. Other elements of the partial order might refer to intermediate results.

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. For example, given a program that outputs an integer, we might decide that we only care whether this integer is even or odd. Then our posets are the even and odd integers, and we will classify operations in the program according

to whether they are evenness-preserving, evenness-inverting, always even, always odd, of uncertain evenness, etc. Most modern compilers use a technique like this (sometimes called abstract interpretation [17]) 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* in nature. For example, 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, and not if there is any uncertainty.

The generality of this approach makes it a kind of “ur type system”: we can discover a large variety of program properties as long as we are willing to accept some uncertainty. Of course, many programmers and language designers prefer to maximize safety, leading to different approaches that trade away some precision (e.g. syntactic type systems such as those in the ML language family [33], [60]). But if static guarantees are not the priority, or if a language considers itself unconcerned with “types”, then we can take the domain-theoretic model as our type system. In some sense, it is our type system whether we like it or not (as implied by [62])!

3.3.1 Monotonicity

When this kind of type system is used, it usually becomes part of the meta-theory of the language: values are “lifted” into an abstract domain, and abstract values are manipulated only by a compiler or other program analysis. One good reason for this is that the abstract values (which we will now interchangeably call “types”) must meet certain requirements in order to be useful. For an abstract interpretation to terminate, its lattice must be of finite height, and all functions of lattice elements must be monotonic (given lattice elements a and

b , we must have $a \subseteq b$ implies $f(a) \subseteq f(b)$ for all f).

Functions of types are not monotonic. `isa(2,3)` is false, but `isa(2,Int)` is true. This provides one way to separate “real types” from set-valued terms: if there is a way to interpret execution over types such that all functions are monotonic in the (alleged) type lattice, then we are really dealing with types. One way to do this is to specify a genuinely multi-valued term language like λ_{\aleph} [31]. In this approach a term like `isa(2,Int)` is interpreted as a union of all results obtained by substituting all possible single values for each sub-term. One cannot help but be impressed by such a language, though it seems prohibitively difficult to actually implement. We opt instead for a simpler approach where `Int` is interpreted as `Type{Int}` (in a near-borrowing of notation from [31] and [10]), and multi-valued terms exist only statically within a separate maximum-fixed-point evaluator.

3.4 Subtyping

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

Chapter 4

The Julia approach

4.1 Core calculus and data model

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

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

The new construct resembles the Dynamic constructor in past work on dynamic typing [2]. The main difference is that the tag is determined by evaluating an expression. Why? The flippant answer is that we are already giving up decidable type checking, so we might as well. But what this really means is that constructing types is part of programming. This can be used to request specific kinds of objects from an API, or used as an implementation detail

to tell the compiler which object properties to track. In practice, `new` is always wrapped in a constructor function, so the programmer defining a data type is the only one who actually instantiates it. 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. However, tags themselves (the first argument to `new`) can be constructed anywhere using nested applications of the syntax $T\{\dots\}$. Tags are a specific subset of data values whose tag is the built-in value `Tag`. Once so constructed, a tag can be passed to a constructor (which is just a generic function) to obtain an instance if desired.

Constants are pre-built tagged values.

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

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

The last item is the special abstract varargs tuple type.

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

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

where \textit{bit}^n represents a string of n bits, and \textit{ref} represents a reference to a tagged data value. Data may be declared mutable, in which case its representation is implicitly wrapped in a mutable cell. A built-in primitive equality operator `==` is provided, based on *egal* [5]

(mutable objects are compared by address, and immutable objects are compared by directly comparing both the tag and data parts bit-for-bit, and recurring through references to other immutable objects).

Functions are generally applied to more than one argument. In the application syntax $e_1(e_2)$, e_2 is an implicitly-constructed tuple of all arguments. 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 function to extend is specified by name rather than by value in order to make it easier to syntactically restrict where functions can be extended. This, in turn, allows the language to specify when new method definitions take effect, providing useful windows of time within which methods do not change, allowing programs to be optimized more effectively (and hopefully discouraging abusive and confusing run time method replacements).

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

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

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

The all-important array type is written as `Array{T,N}` where T is an element type and N is a number of dimensions.

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 [27] [12]. 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!”

To avoid the dual traps of “excess power” and divergence, the system we use must have a decidable subtype relation, and must be closed under data-flow operations (meet, join, and widen). It must also lend itself to a reasonable definition of specificity, so that methods can be ordered automatically (a necessary property for composability). These requirements are fairly strict. Beginning with the simplest possible system, we added features as needed either to satisfy the aforementioned closure properties, or to allow us to write method definitions that seemed particularly useful (as it turns out, these two considerations lead to essentially the same features).

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

$$\llbracket \text{Any} \rrbracket = \mathcal{D}$$

$$\llbracket \text{Bottom} \rrbracket = \emptyset$$

$$T \leq S \Leftrightarrow \llbracket T \rrbracket \subseteq \llbracket S \rrbracket$$

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

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

$$\llbracket \text{Name}\{\dots\} \rrbracket = \{x \mid \text{typeof}(x) = \text{Name}\{\dots\}\}$$

$$\llbracket \text{Abstract}\{\dots\} \rrbracket = \bigcup_{\text{super}(T) = \text{Abstract}\{\dots\}} \llbracket T \rrbracket$$

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

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

$$\begin{aligned} \llbracket \text{Tuple}\{P_1, \dots, P_n\} \rrbracket &= \prod_{1 \leq i \leq n} \llbracket P_i \rrbracket \\ \llbracket \text{Tuple}\{\dots, P_n \dots\} \rrbracket, n \geq 1 &= \bigcup_{i \geq n-1} \llbracket \text{Tuple}\{\dots, P_n^i\} \rrbracket \end{aligned}$$

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

The abstract tuple types ending in \dots 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 [41]. This formulation has the advantage that *any* type that is a subtype of a tuple type can be used to express the signature of a method. It also makes the system simpler and more reflective, since subtype queries can be used to ask questions about methods.

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

$$\llbracket \text{Union}\{A, B\} \rrbracket = \llbracket A \rrbracket \cup \llbracket B \rrbracket$$

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 $\text{Union}\{\text{Int32}, \text{UInt32}\}$.

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 $\text{Union}\{T, \text{Bottom}\}$, or $\text{Union}\{\text{Bottom}, \text{Union}\{T, \text{Bottom}\}\}$, 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:

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

This is one of a few reasons that union types are often considered undesirable. When used with type inference, such types can grow without bound, possibly leading to slow or even non-terminating compilation. Their occurrence also typically corresponds to cases that would fail most static type checkers. Yet from the perspectives of both data-flow analysis and method specialization, they are perfectly natural and even essential [17] [34] [65].

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

$$\llbracket \text{UnionAll } L <: T <: U \ A \rrbracket = \bigcup_{L \leq T \leq U} \llbracket A[T/T] \rrbracket$$

This is equivalent to an existential type [11]; for each concrete subtype of it there exists a corresponding T . Anecdotally, programmers often find existential types confusing. We prefer the union interpretation because we are describing sets of values; the notion of “there exists” can be semantically misleading since it sounds like only a single T value might be involved.

4.2.1 Examples

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

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

4.2.2 Type constructors

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

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

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

$$\boxed{
\begin{array}{c}
\frac{{}^B_A X^L, \Gamma \vdash T \leq S}{\Gamma \vdash \exists {}^B_A X. T \leq S} \quad \frac{{}^B_A X^R, \Gamma \vdash T \leq S}{\Gamma \vdash T \leq \exists {}^B_A X. S} \quad \frac{}{\Gamma \vdash X \leq X} \\
\\
\frac{{}^B X^L, \Gamma \vdash B \leq T}{{}^B X^L, \Gamma \vdash X \leq T} \quad \frac{{}_A X^L, \Gamma \vdash T \leq A}{{}_A X^L, \Gamma \vdash T \leq X} \\
\\
\frac{{}^B X^L, {}_A Y^L, \Gamma \vdash B \leq Y \vee X \leq A}{{}^B X^L, {}_A Y^L, \Gamma \vdash X \leq Y} \quad \frac{{}^B_A X^R, Y^R, \Gamma \vdash B \leq A}{{}^B_A X^R, Y^R, \Gamma \vdash X \leq Y} \\
\\
\frac{{}^B_A X^R, \Gamma \vdash A \leq T}{{}_T^B_A X^R, \Gamma \vdash X \leq T} \quad \frac{{}^B_A X^R, \Gamma \vdash T \leq B}{{}_{A \cup T}^B X^R, \Gamma \vdash T \leq X}
\end{array}
}$$

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 .

4.2.3 Subtyping

Deciding subtyping for base types is straightforward: Bottom is a subtype of everything, everything is a subtype of Any, and tuple types are compared component-wise. The invariant 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 $\text{Tuple}\{\text{Union}\{A, B\} \dots\}$ 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 second row of the figure). We implement the there-exists side by narrowing a variable’s bounds (raising the lower bound and lowering the upper bound). The relation holds as long as the bounds remain consistent (i.e. lower bound \leq upper bound). The algorithm assumes that all input types are well-formed, which includes variable lower bounds always being less than or equal to upper bounds.

However, at this point (starting in the third row, second column) the algorithm appears asymmetric. This is a result of exploiting the lack of contravariant constructors. No contravariance means that every time a right-side variable appears on the *left* side of a comparison, it must be because it occurs in invariant position, and the steps outlined in the first paragraph of this section have “flipped” the order (comparing both $B \leq C$ 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 bottom-left rule in the figure to update X ’s upper bound to $B \cap T$. But because of invariance, $T \leq B$ has already been checked by the bottom-right rule in the figure. Therefore $B \cap T = T$. This is the reason the “forward” direction of the comparison needs to be checked first: otherwise, we would have updated B to equal T already and the $T \leq B$ comparison would become vacuous. Alternatively, we could actually compute $B \cap T$. However there is reason to suspect that serious trouble lies that way. We would need 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, very likely undecidable.

These subtyping rules are very likely Π_2^P -hard. Checking a subtype relation with unions requires checking that for all choices on the left, there exists a choice on the right that makes

the relation hold. This matches the quantifier structure of 2-TQBF problems of the form $\forall x_i. \exists y_i. F$ where F is a boolean formula. If the formula is rewritten in conjunctive normal form, it corresponds to subtype checking between two tuple types, where the relation must hold for each pair of corresponding types. Now use a type $N\{x\}$ to represent $\neg x$. The clause $(x_i \vee y_i)$ can be translated to $x_i <: \text{Union}\{N\{y_i\}, \text{True}\}$ (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.

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:

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

It is not fooled by redundant type variables:

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

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

$$\begin{aligned} \text{Tuple}\{\text{Float32}, \text{Array}\{\text{Float32}, 1\}\} \leq \\ \exists T <: \text{Real} . \exists S <: \text{AbstractArray}\{T, 1\} . \text{Tuple}\{T, S\} \end{aligned}$$

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

$$\exists T . \text{Pair}\{T, T\} < \exists T . \exists S . \text{Pair}\{T, S\}$$

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

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

And the algorithm understands that tuples distribute over unions:

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

4.2.4 Type system variants

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

- Structurally-subtyped records
- μ -recursive types (regular trees)
- Regular types (allowing \dots 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

4.3 Dispatch mechanism

Julia’s dispatch system strongly resembles the multi-method systems found in some object-oriented languages. 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`.¹

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. 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. 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.

this can work especially well for tuple types. by writing `(1, 2.0)` you immediately obtain an efficient “struct” type. cases like protocol compilers require generating code in advance. this generates it on the fly, but can pick up and reuse any cached code that might happen to exist for a particular tuple type.

4.3.2 Specificity

tuples: if an elt of a is more specific than its corresponding elt in b, and no elt of b is more specific than its corresponding elt in a.

this is essentially the same specificity rule used by Dylan [64] for argument lists. Julia generalizes this to tuple types; it is applied recursively for tuple types wherever they occur.

a vararg type is less specific if its corresponding type is equal

Union 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.

¹Our implementation of Julia does not yet have syntax for such methods.

non-union a is more specific than union b if it is more specific than some element

a tag type is more specific than a tag type strictly above it in the hierarchy regardless of parameters (this embeds a moderate amount of “class based” dispatch, compatible with programmer intuition)

a non-variable is more specific than a variable

4.3.3 Ambiguities

our experience: there are lots. e.g. Images and DataFrames

4.3.4 Constructors

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 most 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 related in some standard way. 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.

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
```

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 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 provides a good balance. 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 function of this kind is *guaranteed* to produce the same results for $2+2$ 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 [20] is reading a schema from a database in order to generate code for it. 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.

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 [61] [21] [11] [13]. Since an ordinary function has an arrow type $A \rightarrow B$ 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 $(\text{Float64} \rightarrow \text{Float64}) \cap (\text{Vector} \rightarrow \text{Vector})$. The intuition is that this `sin` function can be used both where a $\text{Float64} \rightarrow \text{Float64}$ function is expected and where a $\text{Vector} \rightarrow \text{Vector}$ 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 immitate the types of higher-order functions in traditional statically-typed functional languages. For example, we might wish

to write `map` as follows:

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

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

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

4.6.2 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 → Int` definition. Naturally, `f` is not applicable to `h`. However, given the call site `f(h)`, we are forced to conclude that `f` might be called with a function of type $(\text{Int} \rightarrow \text{Int}) \cap (\text{String} \rightarrow \text{String})$, since in general `Int → 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

	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.

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 `Any → 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 is 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 $\mathbb{R} \rightarrow \mathbb{R}$, 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 accomodates this approach with a small adjustment to the evaluation rules: in the application syntax $e_1(e_2)$ when e_1 is not a function, evaluate `call(e_1, e_2)` instead, where `call` is a particular generic function known to the system.

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

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

call{N}(p::Polynomial{N}, x) = sum([p.coeffs[i+1]*x^i for i=0:N])
```

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.

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.

It is worth examining the differences and tradeoffs between nominal functions and “real” arrow types.

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 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


```

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.

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 [8] 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}`.

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 studied in the context of those languages.

4.7 Performance model

4.7.1 Type inference

4.7.2 Specialization

it might be thought that julia achieves performance through a clever compiler that uses “every trick in the book”. at risk of being overly self-deprecating, this is not the case. the performance we manage to get derives from the language design, which enables a style of library implementation such that applying straightforward, known optimizations yields good performance.

Multiple dispatch is used heavily throughout the Julia ecosystem. To quantify this statement, we use the following metrics for evaluating the extent of multiple dispatch [54]:

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

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) [54]. The “Julia Operators” row describes 47 functions with special syntax (binary operators, indexing, and concatenation).

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 is focused on technical computing, and so is likely to have a 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

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

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

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

“Diagonal” dispatch lets us improve the definition:

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

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

5.2 Numeric types and embeddings

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

Human beings happen to be good at equating and moving between isomorphic sets, so it is easy to imagine that the reals and complexes with zero imaginary part are one and the same. But a computer forces us to be specific, and admit that a real number is not complex, and a complex number is not real. And yet the close relationship between them is too compelling not to model in a computer somehow. Here we have a numerical analog to the famous “circle and ellipse” problem in object-oriented programming [16]: 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 [59] that generic binary operators should have “key variants” where the arguments are of the same type.

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

```

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

```

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

- Each combination of types only needs to be defined in one order
- It falls back to join
- It must prevent promotion above a certain point to avoid circularity

```
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)

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

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, people have implemented types for interval arithmetic, dual and hyper-dual numbers for computing first and second derivatives, and finite fields.

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

A problem in the design of image processing libraries was solved by defining a new kind of fixed-point type [36]. 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.

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.

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).

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 [30]. 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 [38] uses essentially the same approach as Boost multi-array:

```
instance Slice s1 => Slice (s1 :: Int) where
    sliceOfFull (fsl :: _) (ssl :: _) = sliceOfFull fsl ssl

instance Slice s1 => Slice (s1 :: 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 [7]. If a multiple dispatch system supports variadic functions and argument “splatting” (the ability to pass a structure of n values as n separate arguments to a function), then indexing behavior can be defined as method signatures.

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

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

```
# drop dimensions indexed with scalars
```

```

index_shape() = ()
index_shape(i::Real, I...) = index_shape(I...)
index_shape(i, I...) = tuple(length(i), index_shape(I...)...)

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

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

```

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

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

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

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

5.4 Array views

5.5 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. [19]:

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

Multiple dispatch on special matrices

29 LAPACK types via composition of 9 types, issue 8240

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

qrfact computes the type of the result needed:

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

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

5.6 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 tradeoff 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 [24] 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 m2, SIQuantity{Int64,2,0,0,0,0,0,0})  
  
julia> prod([2m,3m,4m]), typeof(prod([2m,3m,4m]))  
(24 m3, 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 tradeoff a Julia programmer accepts.

5.7 Boundary-element method

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

Recent work [58] managed a more general solution, using Mathematica to generate C++ code for different cases. This worked well, but was difficult to implement and the resulting system is difficult to use. We see the familiar pattern of using multiple languages and code-generation techniques, with coordination of the overall process done either manually or with ad-hoc scripts. To polish the implementation for use as a practical library, a likely next step would be to add a Python interface, adding yet another layer of complexity.

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 [TODO refs], which leads to a matrix L with entries $L_{mn} = \langle b_m, \mathcal{L}b_n \rangle = \langle b_m, b_n \rangle_{\mathcal{L}}$ where $\langle \cdot, \cdot \rangle$ denotes some inner product (e.g. $\langle u, v \rangle = \int uv$ is typical) and $\langle \cdot, \cdot \rangle_{\mathcal{L}}$ is the *bilinear form* of the problem. Computing these matrix elements is known as the matrix *assembly* step, and its performance is a crucial concern for solving partial differential equations (PDEs) and integral equations (IEs).

Singular assembly for integral operators

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) [TODO ref], and involve unknowns on a surface. The analogue of the FEM discretization is then a boundary element method (BEM) [TODO ref], which discretizes a surface into elements (e.g. triangles), with basis functions that are low-order polynomials defined 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 [TODO refs] and similar schemes, but even in this case the diagonal L_{mm} and the entries L_{mn} for adjacent elements must typically be computed explicitly. (Moreover, these are the integrals in which the K singularity is present.)

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

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 [58] 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  $r^p$ ,  $p < 0$ 
abstract APowerLaw{p} <: AbstractKernel

#  $r^p$  power law
type PowerLaw{p} <: APowerLaw{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 [40] 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:

```

type Helmholtz{k} <: APowerLaw{-1}; end #  $e^{ikr}/4\pi r$ 

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

# magnetic field integral equation
type MFIE{k} <: AbstractKernel; end #  $(ikr - 1) * e^{ikr}/4\pi r^3$ 

```

```

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`. However, each kernel requires a large amount of up-front work that cannot be done efficiently at run time. Julia's staged functions solve this problem.

```

@generated function call{p,P<:APowerLaw{p},n}(K::FirstIntegral{P,n},
                                                X::Real)

    quote
        X <= 0 && throw(DomainError())
        t = ....
        @evalpoly(t, ...coefs...)
    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.

pseudocode for how this is used:

```

# from paper figure 1
Fct = [FirstIntegral{K,i}() for i=1:n]
Pct = [some polynomials...]
for trianglepair in mesh
    i, j = indexes based on triangles
    if common(trianglepair)
        M[i,j] = integrate(y->sum([Fct[p](y)*Pct[p](y) for p=1:n]),
                           0, 1)
    elseif commonedge(trianglepair)
        # a 2d integral
    elseif commonvertex(trianglepair)
        # a 3d integral
    end
end

```

```

    else
        # separate triangles
    end
end
end

```

5.8 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 [25], which allows users to specify problems using a high-level mathematical syntax. Here is an example model of a “knapsack” problem (from [44]) 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 [44], 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

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 5.1: Performance (time in seconds) of several linear programming tools for generating models in MPS format (excerpted from [44]). L is the number of locations solved for in a facility location problem.

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.

5.9 Beating the incumbents

- erfinv and digamma using horner macro

- randn beating matlab

@evalpoly macro has separate cases for real and complex in order to automatically take advantage of a subtle algorithm [39]. A macro is perfect for generating the neccessary code, however it lacks the type information needed to select between the two cases. In Julia, it can generate a branch with a type check, and rely on the unused case being removed by automatic specialization. (A generic function with two definitions could be generated instead, but in high-performance programming the “force inlining” behavior of macros is welcome.)

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

grisu: 6kLOC to 1kLOC (PR 7291)

Chapter 6

Conclusion

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

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

julia encourages lattice-theoretic execution models to become integral parts of languages. dispatch was an especially natural way to do this, are there others?

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

We have speculated about whether future languages will be able to do away with this distinction. One approach is λ_{κ} . We have also speculated that this could be done using types based on non-well-founded set theory, combining the subset-of and element-of relations using self-containing sets. We are not yet sure what a practical language based on this idea might look like.

There are several key aspects of performance programming that our design does not directly address, e.g. storage and in-place optimizations.

future work: product domains

how to incorporate finer types more smoothly

Appendix A

Subtyping algorithm

```
abstract Ty

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

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

type UnionT <: Ty
  a; b
end

type Var
  lb; ub
end

type UnionAllT <: Ty
  var::Var
  T
end

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

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

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

```

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

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

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

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

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

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

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

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

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

```

```

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

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

        found = issub(x, y, env)

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

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

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

```

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

```
function issub(a::TagT, b::TagT, env)
  a === b && return true
  b === AnyT && return true
  a === AnyT && return false
  if a.name !== b.name
    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  $\forall a, b . a <: b$ 
```



```

        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  $\forall b . b <: a$ 
    !issub(bb.lb, a, env) && return false
    # for contravariance we would need to compute a meet here, but
    # because of invariance  $bb.ub \sqcap a == a$  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  $\forall b . b >: a$ 
    !issub(a, bb.ub, env) && return false
    bb.lb = join(bb.lb, a, env)
    return true
end

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

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

```

Bibliography

- [1] *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., DYBVIK, 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] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] BRUCE, K., CARDELLI, L., CASTAGNA, G., LEAVENS, G. T., AND PIERCE, B. On binary methods. *Theor. Pract. Object Syst.* 1, 3 (Dec. 1995), 221–242.

- [10] CARDELLI, L. *A Polymorphic λ -calculus with Type:Type*. Digital Systems Research Center, 1986.
- [11] CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.* 17, 4 (Dec. 1985), 471–523.
- [12] 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.
- [13] CASTAGNA, G., GHELLI, G., AND LONGO, G. A calculus for overloaded functions with subtyping. *Inf. Comput.* 117, 1 (Feb. 1995), 115–135.
- [14] 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.
- [15] 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.
- [16] CLINE, M. P., AND LOMOW, G. A. *C++ FAQs*. Addison-Wesley, 1995.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] 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.

- [21] 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.
- [22] 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.
- [23] FALKOFF, A. D., AND IVERSON, K. E. The design of APL. *SIGAPL APL Quote Quad* 6 (April 1975), 5–14.
- [24] FISCHER, K. SI units package, 2014. <https://github.com/Keno/SIUnits.jl>.
- [25] FOURER, R., GAY, D. M., AND KERNIGHAN, B. *AMPL*, vol. 119. Boyd & Fraser, 1993.
- [26] 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”.
- [27] 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.
- [28] 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.
- [29] 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 Programming, Systems, Languages, and Applications* (New York, NY, USA, 2003), OOPSLA '03, ACM, pp. 115–134.
- [30] GARCIA, R., AND LUMSDAINE, A. MultiArray: a C++ library for generic programming with arrays. *Software: Practice and Experience* 35, 2 (2005), 159–188.
- [31] GLEW, N., SWEENEY, T., AND PETERSEN, L. A multivalued language with a dependent type system. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming* (New York, NY, USA, 2013), DTP '13, ACM, pp. 25–36.
- [32] GOMEZ, C., Ed. *Engineering and Scientific Computing With Scilab*. Birkhäuser, 1999.

- [33] HINDLEY, R. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* (1969), 29–60.
- [34] 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.
- [35] IHAKA, R., AND GENTLEMAN, R. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5 (1996), 299–314.
- [36] JULIA. Parametric color types, 2014. <http://github.com/JuliaLang/Color.jl/issues/42>.
- [37] KELL, S. In search of types. *Proc. 2014 ACM Int. Symp. New Ideas, New Paradig. Reflections Program. Softw. - Onward! '14* (2014), 227–241.
- [38] 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.
- [39] KNUTH, D. E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. Addison-Wesley, 1969.
- [40] LATNER, 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).
- [41] LEAVENS, G. T., AND MILLSTEIN, T. D. Multiple dispatch as dispatch on tuples. *ACM SIGPLAN Not.* 33, 10 (Oct. 1998), 374–387.
- [42] 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.
- [43] LOGG, A., ØLGAARD, K. B., ROGNES, M. E., AND WELLS, G. N. *FFC: the FEniCS Form Compiler*. Springer, 2012, ch. 11.
- [44] LUBIN, M., AND DUNNING, I. Computing in operations research using julia. *INFORMS Journal on Computing* 27, 2 (2015), 238–248.
- [45] MATHEMATICA. <http://www.mathematica.com>.
- [46] MATHWORKS, INC. *MATLAB Manual: linspace*, r2014b ed. Natick, MA.
- [47] MATHWORKS, INC. *MATLAB Manual: mldivide*, r2014b ed. Natick, MA.

- [48] MATLAB. <http://www.mathworks.com>.
- [49] MERRILL, J. Julia vs mathematica (wolfram language): high-level features, 2014. <https://groups.google.com/d/topic/julia-users/EQH5LJZqe8k/discussion>.
- [50] MILLER, E. Why I'm betting on julia, 2014. <http://www.evanmiller.org/why-im-betting-on-julia.html>.
- [51] 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.
- [52] MORRIS, J. H. J. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.
- [53] MURPHY, M. Octave: A free, high-level language for mathematics. *Linux J.* 1997 (July 1997).
- [54] MUSCHEVICI, R., POTANIN, A., TEMPERO, E., AND NOBLE, J. Multiple dispatch in practice. *SIGPLAN Not.* 43 (October 2008), 563–582.
- [55] PIERCE, B. Bounded quantification is undecidable. *Information and Computation* 112, 1 (1994), 131 – 165.
- [56] 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).
- [57] RATHGEBER, F., MARKALL, G., MITCHELL, L., LORIENT, 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.
- [58] 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.
- [59] 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.
- [60] ROBIN, AND MILNER. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17, 3 (1978), 348 – 375.
- [61] RONCHI DELLA ROCCA, S. Principal type scheme and unification for intersection type discipline. *Theor. Comput. Sci.* 59, 1-2 (July 1988), 181–209.

- [62] SCOTT, D. Data types as lattices. *Siam Journal on computing* 5, 3 (1976), 522–587.
- [63] SCOTT, D., AND STRACHEY, C. *Toward a mathematical semantics for computer languages*, vol. 1. Oxford University Computing Laboratory, Programming Research Group, 1971.
- [64] 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.
- [65] 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.
- [66] SNIR, M. *MPI—the Complete Reference: The MPI core*, vol. 1. MIT press, 1998.
- [67] STEELE, G. L. *Common LISP: the language*. Digital press, 1990, ch. 4.
- [68] 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.
- [69] VAN DER WALT, S., COLBERT, S. C., AND VAROQUAUX, G. The NumPy array: a structure for efficient numerical computation. *CoRR abs/1102.1523* (2011).
- [70] 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.
- [71] WADLER, P. The expression problem. *Java-genericity mailing list* (1998).