TERRA: SIMPLIFYING HIGH-PERFORMANCE PROGRAMMING USING MULTI-STAGE PROGRAMMING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Zachary DeVito

December 2014

ABSTRACT

Modern high-performance computers are heterogeneous machines that contain a mixture of multi-core CPUs, GPUs, interconnects, and other custom accelerators. The composition of individual machines varies and will change over time, so it is important that applications written for these machines are portable. One way to write portable high-performance applications is to abstract the problem at a higher-level and use approaches such as auto-tuners or domain-specific languages (DSLs) to *compile* the abstraction into high-performance code using domain knowledge. These approaches require that the application generate code programmatically. However, current state-of-the-art high-performance languages lack support for generating code, making it difficult to design these programs.

This thesis presents a novel two-language design for generating high-performance code programmatically. We leverage a high-level language, Lua, to stage the execution of a new low-level language, Terra. Programmers can implement program analyses and transformations in Lua, but use built-in constructs to generate and run high-performance Terra code. This design reflects how programs like DSLs are often created in practice but applies principled techniques to make them easier to create. We adapt techniques from multi-stage programming to work within the two-language design. In particular, shared lexical scope makes it easy to generate code, but separate execution enables the programmer to control performance. Furthermore, we show how adding staged compilation to the typechecking process of Terra allows for the creation of high-performance types. This typechecking process combines meta-object protocols frequently used in dynamically-typed languages with staged programming to produce flexible but high-performance types.

We evaluate the design presented in this thesis by implementing a number of example programs using Lua and Terra in areas where performance is critical such as linear algebra, image processing, probabilistic programming, serialization, and dynamic assembly. We show that these applications, which previously required many different languages and technologies, can be implemented concisely using just Lua and Terra. They perform as well as or better than their equivalents written using existing technologies. In some cases, we found that the added expressiveness of using our approach made it feasible to implement more aggressive optimizations, enabling the program to perform up to an order of magnitude better than existing approaches.

ACKNOWLEDGMENTS

Creating this dissertation was only possible with the help, suggestions, and support of many people. First, I want to thank my advisor Pat Hanrahan for always encouraging me to work on the problems that I found most interesting, and giving me scarily accurate advice well before I realized that I needed it. Frequently there were times that the solution to a problem that I was working on was something Pat mentioned off-hand six months earlier.

I am also grateful to Alex Aiken for his advice over the years on the many projects I worked on including the Terra and Liszt languages. As my interests turned to programming languages, Alex helped me navigate the community, always asking on-point questions that helped focus the work. I also want to thank my other Committee members — Kunle Olukotun, Jan Vitek, and Eric Darve — for their advice and for many fruitful discussions about how to design high-performance domain-specific languages.

My work would not have been possible without the help of many other students in the lab at Stanford. I am grateful to James Hegarty for participating in endless discussions about programming language design, and for using Terra even at a very early stage, and to Daniel Ritchie and Matt Fisher for their help in crafting the design of Exotypes.

I started working on the Liszt programming language only a few months after I came to Stanford. The project has continued up to this day, accumulating a large number of people who have helped out over the years. I am grateful for this help, especially to Niels Joubert, who spent long hours getting Liszt to run on large clusters and GPUs, and to Crystal Lemire, Gilbert Bernstein, and Chinmayee Shah who are now continuing work on the project.

I also want to thank my academic "siblings" Mike Bauer, Eric Schkufza, Sharon Lin, and Lingfeng Yang who helped me navigate the academic process, and encouraged me to keep going on problems even when I thought it was not worth it. I am also grateful to the rest of Pat’s lab and other students I interacted with while at Stanford. Their diverse interests, enthusiasm for problem solving, and inquisitiveness meant that meetings never had a dull moment.

Finally, I would like to thank my mom and dad for their support and encouragement through this process. I wouldn’t have been able to do it without you.

CONTENTS

ABSTRACT iv

ACKNOWLEDGMENTS vi

1 INTRODUCTION 1

1.1 Contributions 4

1.2 Thesis outline 6

2 BACKGROUND 7

2.1 Practical designs of libraries that do code generation 7

2.1.1 Ad hoc templates and scripts 8

2.1.2 Source-to-source autotuners 9

2.1.3 Statically-compiled DSLs 11

2.1.4 Dynamically-compiled DSLs 14

2.1.5 Summary 16

2.2 Existing techniques for code generation 16

2.2.1 LISP 17

2.2.2 Partial evaluation 19

2.2.3 Multi-stage programming 21

2.3 Conclusion 23

3 GENERATING CODE 25

3.1 Writing code in Terra 26

3.2 Multi-stage programming in Terra 30

3.3 Writing optimizations using staging 32

3.4 Generating arbitrary code using staging. 33
3.5 Design decisions for code generation 36
3.5.1 Shared lexical environment 36
3.5.2 Hygienic code generation 37
3.5.3 Separate evaluation of Terra code 38
3.5.4 Eager specialization with lazy typechecking 39

4 FORMALIZING CODE GENERATION WITH TERRA CORE 41
4.1 Terra Core 41
4.2 Design decisions as reflected in Terra Core 46
4.2.1 Terra entities are first-class 46
4.2.2 Hygiene and shared lexical environment 47
4.2.3 Eager specialization with lazy typechecking 48
4.2.4 Separate evaluation of Terra code 50

5 BUILDING DSLS 51
5.1 Tuning DGEMM 51
5.2 Darkroom: A stencil DSL for images 55
5.3 Liszt: A DSL for physical simulation 57
5.4 Quicksand: A DSL for probabilistic programming 59

6 GENERATING TYPES WITH EXOTYPES 63
6.1 Meta-object protocols 65
6.2 Exotypes interface 66
6.3 Example: Array(T) 70
6.4 Composability 72
6.5 Related Work 74

7 FORMALIZATION OF EXOTYPES 76

8 BUILDING LIBRARIES WITH EXOTYPES 83
8.1 Class systems 83
8.2 Data layout 85

8.3 Serialization 87
8.4 Dynamic x86 assembly 89
8.5 Automatic differentiation 92

9 EMBEDDING CUSTOM DSLS IN LUA 95
9.1 Current approaches to embedding languages 95
9.2 Our language extension interface 98
9.3 Using language extensions 100
9.3.1 Darkroom: Lua-based meta-programming with custom syntax . . . 100
9.3.2 Liszt: Custom syntax with a language extension API 102
9.4 Comparison to other approaches to syntax extension 103

10 INTEROPERABILITY 105
10.1 With existing applications 105
10.1.1 Embedding as a runtime 106
10.1.2 Zero-overhead compatibility with C 107
10.2 With Lua 111
10.3 With DSLs 115

11 IMPLEMENTATION 117

12 DISCUSSION 121
12.1 Lessons learned from building DSLs 121
12.2 Software development as a library 123
12.3 Statically-typed languages as heterogeneous staged programming 124
12.4 A middle ground for static and dynamic typing 126
12.5 Applying dynamic code generation to scientific computing 128
12.6 Using high-performance staged programming to improve compiler performance 128

13 CONCLUSION 130

List of Figures

2.1 Example Liszt code 12
2.2 Generating code with LLVM 15
2.3 Macro hygiene 18
2.4 Multi-stage programming operators 22

3.1 Phases of evaluation of a Lua-Terra program. 30
3.2 An example function that compiles a simple IR for image processing code into a Terra function 35
3.3 A simple autotuner for the image processing language. 36

4.1 The rules $\xrightarrow{L}$ for evaluating Lua expressions. 44
4.2 The rules $\xrightarrow{S}$ for specializing Terra expressions. 45
4.3 The rules $\xrightarrow{T}$ for evaluating Terra expressions. 45
4.4 Typing rules for references to Terra functions. 45

5.1 Generator for matrix-multiply kernel 52
5.2 Matrix multiply performance 53
5.3 Darkroom compared to C 54
5.4 Speed using Darkroom 55
5.5 A higher-order Lua function that wraps a Terra function for use in MCMC inference. 60
5.6 Quicksand performance 62

6.1 Extension of Terra evaluation for exotypes 67
6.2 Example exotype properties 67
6.3 Embedding Objective-C objects in Terra using exotypes. 70

6.4 An implementation of the array constructor in Lua 71
6.5 An implementation of the array constructor using Exotypes in Terra 72

7.1 Typing rules 77
7.2 ExoTerra Evaluation 78
7.3 The rules for evaluating ExoLua 78

8.1 Example code that uses our class system interface 84
8.2 Data layout performance 86
8.3 A generator for serialization code 88
8.4 Throughput of scene graph serialization 88
8.5 Assembler throughput 89
8.6 Example using our dynamic assembler 92
8.7 Automatic differentiation performance 94

9.1 Examples of different techniques for implementing the frontend of a DSL . 96
9.2 An example language extension 99
9.3 Embedding Darkroom in Lua 101

10.1 Embedding Lua in C 106
10.2 Embedding Terra in C 107
10.3 Correspondence between Terra and C types 108
10.4 Calling external functions form different languages. 110
10.5 Converting Lua values to Terra values. 112
10.6 Inferring the types of Lua values 113
10.7 Converting Terra values to Lua values 113

11.1 Overview of Terra implementation 118

CHAPTER 1
INTRODUCTION

To achieve higher performance and power efficiency, modern computers are composed of a heterogeneous mix of cores and accelerators, each specialized to achieve high efficiency for particular tasks. A modern laptop might have multiple CPU cores, whose out-of-order execution and large caches make them ideal for serial tasks where low-latency is important. On the same chip, there is an integrated GPU for high-efficiency processing of throughput-oriented tasks such as real-time rendering. For high-performance graphics processing an additional discrete GPU may be included as well. Similar complexity is found at both the small and large scale. Mobile phones use systems on a chip that contain many specialized cores for tasks such as image processing, while large supercomputers often contain a mixture of CPUs, GPUs, and high-speed interconnects across disparate memory spaces.

To write high-performance and high-efficiency applications, a programmer must take advantage of all these devices. While an application can be written for one set of hardware, the amount of variety across modern computers leads to an exponential number of potential configurations. Furthermore, the particular hardware is likely to change over time, making effort spent optimizing a single design obsolete in a few years. Some widely used applications might be worth optimizing in this way, but most programs will not be able to take advantage of the higher-efficiency hardware.

We want to create applications that are portable across different hardware and programming models. One way to achieve this portability is by raising the level of abstraction at which applications are programmed. Rather than write a program as a stream of instructions for a particular piece of hardware, code can be written at the level of the problem. Languages that support this level of programming are referred to as domain-specific languages, or DSLs.

A compiler specialized for a particular domain is used to transform high-level code into actual machine code for different architectures. Since problems are expressed at the level of a domain, experts in that domain can automatically apply best-practice techniques to get the problem to run on different hardware. Widely used domain-specific languages for graphics programming such as OpenGL/DirectX have made it possible to run applications across different GPUs without programming to a specific architecture. And recent work such as OptiML for machine learning [7], or Liszt [22] for physical computation on meshes have shown that the DSL approach can produce code that takes advantage of heterogeneous hardware, while still being portable to different configurations or new machines.

Expressing problems as transformations on higher-level data can also be used to create efficient and portable libraries more easily. This approach has been proven for problems such as Fourier transforms and linear algebra, with libraries such as SPIRAL [60], FFTW [30], and ATLAS [77] which internally use domain-specific transformations to optimize performance. These libraries often rely on auto-tuning — trying a lot of possible optimizations on a particular architecture — to find the best one. And although their APIs look like any other library, internally these libraries, like DSLs, need to actually generate and run code as part of their execution. They are often referred to as *active* libraries to distinguish them from libraries that do no code generation.

Despite the promise of DSLs and active libraries, their use is limited to a few large domains because it is currently difficult to actually create an active library or DSL. One reason for this difficulty is that these designs need to generate code programmatically. This need to generate or *meta-program* code makes their architecture fundamentally different from typical libraries. To get peak performance on modern hardware, current programmers use languages such as C++ or CUDA where the level of abstraction is only slightly above the level of the hardware. These languages give the programmer a lot of control over performance, but they lack facilities for doing code generation, forcing designers of DSLs to use ad hoc approaches. The complexity of such designs limits adoption to only a few experts and makes it harder to integrate the DSL into existing applications.

To make it easier to write portable high-performance applications, we need the ability to generate code in high-performance languages. But adding support for code generation to these languages is difficult because there is a fundamental tension between features that make meta-programming easy and features that give the programmer control over performance. To make code generation and code transformations easier, it is desirable to have high-level features such as an object system for representing trees and graphs of program representations, first-class functions to apply transformations, and garbage collection to manage these data structures. We further need a way for the programmer to pass these data-structures to a library that will actually compile the result. But to provide control over performance, we need to support low-level features. The language should have semantics that are close to the actual instruction set of the machine, which includes the ability to manage the layout of memory to optimize memory performance. The language also needs to be simple enough that it can easily run on smaller cores, like those found on GPUs or embedded devices.

A single language that tries to balance the concerns of both code generation and low-level control over performance would struggle to combine both sets of features, and could end up being a poor fit for either problem. Instead, this thesis proposes using a *two-language* design that separates the concerns of code generation and control over performance. We use a high-level language to write program analyses and generate code, and a low-level language to write high-performance code. The two-language design reflects the way many DSLs and auto-tuners are designed in practice, but applies principled techniques to make creating such designs easier.

For the two-language design to work, it is important that the languages interoperate well. To support code generation, the high-level language must have the ability to meta-program the low-level one. We use an existing language, Lua [39], as our high-level language since it was designed to be used as an embedded language from C. Lua has well thought out solutions to problems that arise when a low-level language is coupled to a high-level one, such as the management of Lua object lifetimes, the representation of low-level data in Lua, and how functions are evaluated across both languages. However, existing low-level languages are not suited to working with high-level languages or being the target of code generation. To fill this gap, this thesis presents a new low-level language Terra. Terra has semantics that are similar to C but it is designed to be meta-programmed and evaluated from Lua.

1.1 Contributions

The proposed two-language design can simplify the process of writing portable high-performance applications by making it easier to develop programs that require code generation. This thesis will present the contributions necessary to make this design work, and several case studies that evaluate its ability to make the creation of high-performance programs easier.

- We propose a two-language design for generating high-performance code that uses Lua, a high-level language to meta-program Terra, a new low-level systems programming language. Meta-programming is accomplished by adapting techniques from multi-stage programming to work in the context of a two-language system. In particular we show how shared lexical scope makes it easy to generate code, but separate evaluation enables the programmer to control performance.

- We evaluate our two-language design by creating example DSLs and autotuners in the areas of linear algebra, image processing, physical simulation, and probabilistic programming. We show that these applications, which previously required the use of many languages and technologies, can be architected entirely in Lua and Terra but still achieve high-performance. Our auto-tuner for matrix multiply performs within 20% of ATLAS [77] but uses fewer than 200 lines of Terra code. Our image processing language performs 2.3x faster than hand-written C and is comparable to the performance of Halide [61], an existing image processing language. Our implementation of a probabilistic programming language runs 5 times faster than existing implementations.

- To support the creation of robust active libraries, we also propose a new system for the runtime generation of high-performance types, called *exotypes*. We leverage the two-language design of Lua and Terra to blend the flexibility of types found in dynamically-typed programming languages with the speed of types from statically-typed low-level languages. To do this, we combine meta-object protocols, a technique popular in dynamically-typed languages, with traditional staged programming techniques. This process runs during Terra's typechecking phase.

- We show how using exotypes makes it possible to create fast, concise, and composable libraries of types. We evaluate the use of exotypes in several performance-critical scenarios including serialization, dynamic assembly, automatic differentiation, and data structure layout. In the scenarios we evaluate, we show how we can achieve expressiveness similar to libraries written in dynamically-typed languages while matching the performance of existing implementations written in statically-typed languages. The added expressiveness makes it feasible to implement aggressive optimizations that were not attempted in existing static languages. Our serialization library is 11 times faster than Kryo (a fast Java serialization library). Our dynamic x86 assembler can assemble templates of assembly 3–20 times faster than the assembler in Google Chrome.

- To make the interactions between Lua and Terra precise, we provide formal semantics for their evaluation, and for the evaluation of exotypes that occurs during Terra's typechecking. These semantics illustrate the issues and design choices that arise when using one language to meta-program another.

- We present a method of extending Lua's syntax that allows user-defined languages to be embedded in Lua using the same approach we used to embed Terra. We show how this design allows the creation of DSLs with custom syntax, while also providing the benefits of shared lexical scoping and separate evaluation to DSLs.

- We describe the design choices used in our two-language design that make it easy to integrate it into existing applications and have multiple languages interoperate. We discuss how these decisions made some of our example applications easier to architect.

Terra is open source and available online at `terralang.org`. In addition to our implementation, we also provide a "getting started" guide, API reference, and example code. Our implementation of Terra is continually being updated with the intention of supporting the ongoing development of DSLs, including those presented in later chapters.

1.2 Thesis outline

Chapter 2 describes the challenges that arise when writing high-performance applications like DSLs and autotuners by examining a few designs used in practice and presents related work that proposes principled techniques for code generation.

Chapters 3–5 describe Terra's approach for generating code. Chapter 3 introduces the Terra language and shows how we use multi-stage programming to perform the tasks required to build portable high-performance applications. Chapter 4 describes our formal semantics for this system, and Chapter 5 presents example designs using this system that perform as well as or better than the previous state-of-the-art while being significantly simpler to create. This work is based on results published in [23] with contributions from J. Hegarty, P. Hanrahan, A. Aiken, and J. Vitek.

Chapters 6–8 describe Terra's approach for generating types. Chapter 6 describes our exotypes interface, which combines meta-object protocols with staged programming. Chapter 7 provides formal semantics for the property lookups that occur during typechecking exotypes. Chapter 8 shows example libraries of types built using exotypes that perform better than state-of-the-art libraries while being significantly more concise. This work is based on results published in [24] with contributions from D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan.

Finally, Chapter 9 shows how we can extend the properties of our two-language design to include more languages in one ecosystem using our language extension mechanism. Chapter 10 describes the features of our two-language design that make it easy to integrate into existing applications and have multiple languages interoperate. Chapter 11 provides more details about how we implemented Terra. Chapter 12 concludes with a discussion of the contributions and by presenting some interesting pathways for future work.

Chapter 2
Background

There have been two lines of work on using code generation to get higher performance, *practical* designs for active libraries or DSLs and *research* on language designs and techniques for doing principled code generation. Here we examine some of the previous work in both of these areas. Our goal when creating the two-language design of Terra was to adapt the principled approaches from the literature to the actual patterns of DSL and library design seen in practice.

2.1 Practical designs of libraries that do code generation

For frequently used domains, it is worth the cost of developing a DSL or active library even without code generation tools that make the task easier. We can use already existing tools in these areas to examine how developers currently structure these kinds of applications, and what challenges arise.

The DSLs and autotuners presented here all share similarities in their architecture, and contain similar components:

1. A frontend that is used as input to the problem. For DSLs, this will be a language, but for active libraries this may simply be a particular specialized version of a function call.

2. A set of transformations on the input that use domain knowledge to transform the program into high-performance code. In some systems, these transformations may be

simple, but in others these transformations might occur in many stages and use many different *intermediate representations* (IR).

3. A code generator that takes the last IR produced by the transformations and emits code in an existing language. In the simplest designs, this may just write out strings of a programming language such as C.

4. A runtime library that is used by the generated code. Often it is the case that the generated code will call out to already existing library code, for loading data files or other subtasks that do not benefit from program-specific optimizations.

To illustrate how these components are created, we can look at several different approaches found in practice.

2.1.1 *Ad hoc templates and scripts*

Many libraries that do auto-tuning are built out of ad-hoc scripts that do code generation by invoking offline compilers. A widely used example of this design is *ATLAS*, an autotuner for linear algebra operations supported in BLAS [77]. While it includes support for all BLAS operations, it does most of its optimization on level 3 BLAS routines that are focused on matrix multiply.

The architecture of ATLAS consists of a frontend that describes the desired operations (matrix-matrix multiply, matrix-vector, etc.) and the sets of data types (float, double, complex). From this high-level specification, it will apply transformations to optimize individual operations. For instance, ATLAS breaks down a matrix multiply into smaller operations where the matrices fit into L1 cache. Then, an optimized kernel for L1-sized multiplies is used for each operation. To create this kernel, ATLAS uses autotuning techniques to choose good block sizes, loop unrolling factors, and vector lengths to generate optimized code for the L1-sized kernel. After choosing a set of parameters to try it will generate a C or assembly program using those factors, compile it, and then time it. Once the L1-sized kernel is generated, it is linked with a pre-existing library for handling the blocking and parallelization across multiple cores.

While conceptually simple, it is hard to architect ATLAS using current tools. ATLAS is focused on achieving high-performance, so it targets low-level languages such as C or x86 assembly. These languages provide fine-grained control of performance, making it more likely that ATLAS can find a high-performance solution. However, these languages lack support for code generation, forcing ATLAS to rely on external tools to create and build these programs. The result is a fairly complicated architecture — the frontend consists of Makefiles and scripts that assemble the different BLAS operators. These scripts run in different processes to actually do the code generation. Code generators themselves are written in a mixture of languages. Some are C programs that write text of other C programs, others are templates that use a combination of C-preprocessor macros and a custom templating language to stitch fragments of x86 assembly together. In addition to using many different technologies, there are many different processes running — the top level generator, individual template generators, C compilers, and runtime timing of generated code. Since each has a different memory space, transferring information between processes requires explicit serialization. The combination of different tools communicating across process boundaries makes these ad hoc designs difficult to maintain.

2.1.2 *Source-to-source autotuners*

Autotuners like ATLAS only need to search over a few parameters such as blocking sizes, and loop unrolling factors, making ad hoc techniques based on pre-processors feasible. Other autotuners search over a wider variety of fundamental algorithmic transformations. These systems do more complicated transforms and are often architected as source-to-source programs, where the resulting program is emitted as a string to a file, and a traditional offline compiler is used to compile the result. One area where this approach is popular is for discrete Fourier transforms (DFTs). The simple description of a DFT over a vector of length N has a running time of $O(N^2)$, but can be decomposed into smaller DFTs in a way that results in a *fast* Fourier transform that is $O(N \log N)$. Multiple techniques exist for decomposing DFTs — Cooley-Tukey, Prime-factor, Rader, and Bluestein, among others — and the applicability of each depends upon the factorization of N. So the best approach can be very different across different values of N, and optimized FFT libraries generate different approaches for different values of N.

FFTW [30] is one library that autotunes over different decomposition approaches. It uses a combined offline and online approach to getting high performance. For small sizes it uses pre-generated FFT code that was autotuned offline, which they refer to as *codelets*. This code unrolls most loops and performs optimization over the resulting expressions. Larger sizes are "planned" dynamically for the particular machine they are running on using heuristics that invoke the fast pre-generated code. At this level recursive decompositions are chosen using an interpreter that examines the plan and invokes the right codelets.

The bifurcation between online and offline optimization is reflected in the architecture of FFTW and the technologies it uses. The original version of FFTW performed codelet generation with a compiler written in OCaml, and then emitted C programs which were compiled with an external compiler and timed. The larger FFTs were then planned dynamically by a library written in C and linked into user programs. Like ATLAS, this design required moving data about the transforms through multiple technologies and processes — the OCaml compiler, C compilation and autotuning, and data structures to represent the plan in the FFTW C runtime.

Further work on FFT optimization in the SPIRAL [60] framework gained better performance by decomposing the problem of FFT optimization into more stages. SPIRAL introduces an intermediate language, SPL, that describes decompositions of FFTs. It can be manipulated mathematically to optimize the code. In addition to invoking smaller FFTs, operators in SPL also need to shuffle or permute the data for correct decompositions. For the best memory performance, it is important that these shuffles are not performed in a separate pass over the data but are folded into the indexing of consumers instead. In FFTW this was achieved by unrolling the code and then performing simple straight-line optimizations to precompute most of the indexing. But this approach does not scale to larger sizes where everything is not unrolled. In the SPIRAL framework, SPL is high-level, and does not represent this indexing explicitly. So SPIRAL introduces a lower-level intermediate representation, Σ-SPL, to explicitly represent these indexing operators [29]. At this level it is possible to perform optimizations such as loop merging, and array-index simplifications that were obscured at the level of SPL, but would not typically be optimized by low-level compilers.

The SPIRAL design illustrates a common feature of many of these high-performance optimization libraries, where optimizations are expressed as transformations on *multiple* levels of intermediate representations — SPL, Σ-SPL, and low-level code. The final representation bears little resemblance to the original high-level description.

2.1.3 *Statically-compiled DSLs*

So far we have looked at the design of autotuners, where the input is a description of a function call that we want to optimize. Domain-specific languages are also architected using similar techniques. Liszt [22] is a DSL that we developed for doing physical simulation. Liszt is implemented with a frontend written in Scala, but using cross-compilation to emit C++ code that makes calls into C++ runtimes for clusters, SMPs, or GPUs. Scala was chosen for the frontend, because of its functional language features and its ability to easily declare tree-like datatypes that made it a productive language for writing compiler analyses. However, we needed to emit C++ to achieve high performance in the runtime, and also because Liszt needed to use already existing high-performance interfaces such as MPI or CUDA, which are primarily accessed using the C ecosystem.

We originally developed Liszt as a case study in DSL design, so it is worth looking at what it does in more detail since it can illustrate the issues that arise when designing a DSL using static compilation.

Computation in Liszt is expressed on a mesh of topological elements such as faces and vertices. Work is normally done in parallel across each element of the mesh using information from a local *stencil* of neighboring elements. Physical simulation can be done on a mesh with regular topology (such as a rectilinear grid), where the dependencies between elements can be expressed as affine computations of the element coordinates. Liszt is more general. It supports meshes of arbitrary topology and internally uses graph data structures to track this topology.

This model of computation has large amounts of mostly-parallel work since it can operate over all mesh elements at once. But computation on each element has dependencies on neighboring elements in the stencil. Current state-of-the-art libraries that run these computations on large clusters of machines use an approach based on domain decomposition. Each machine is given a partition of the mesh elements that it is responsible for simulating. Along the border or *halo* of these partitions, the stencil of individual elements will require data from other machines. The management of what data to exchange between partitions

```
// Fields are maps from an element of the mesh to a value.
// The Position field stores a vector of three floats at each vertex.
val Position = FieldWithLabel[Vertex,Float3]("position")
val Temperature = FieldWithConst[Vertex,Float](0.f)
val Flux = FieldWithConst[Vertex,Float](0.f)
val JacobiStep = FieldWithConst[Vertex,Float](0.f)

// for statements operate over the set of mesh elements in parallel
// here we operate over all the vertices in the mesh to set
// initial conditions
for (v <- vertices(mesh)) {
   if (ID(v) == 1)
      Temperature(v) = 1000.0f
   else
      Temperature(v) = 0.0f
}
// Perform Jacobi iterative solve
var i = 0;
while (i < 1000) {
    for (e <- edges(mesh)) {
       // Liszt includes built-in topological operators such as 'head(e)'
       // that look up neighboring topology
       val v1 = head(e)
       val v2 = tail(e)
       // Field values of neighboring topology can be read or written:
       val dP = Position(v2) - Position(v1)
       val dT = Temperature(v2) - Temperature(v1)
       val step = 1.0f/(length(dP))
       // Commutative and associative reduction like +=
       // can be performed in parallel across the mesh:
       Flux(v1) += dT*step
       Flux(v2) -= dT*step
       JacobiStep(v1) += step
       JacobiStep(v2) += step
    }
    for (p <- vertices(mesh)) {
       Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
    }
    for (p <- vertices(mesh)) {
        Flux(p) = 0.f; JacobiStep(p) = 0.f;
    }
    i += 1
}
```

Figure 2.1: This Liszt example calculates the temperature equilibrium due to heat conduction on a grid using Jacobi iteration.

is tricky because it depends on the stencil of the particular algorithm being run *and* the particular topological relationships in the mesh actually being used. Most pre-existing solutions discover the halo region using a hand-written analysis of the topology of the mesh. This approach is prone to error since this analysis has to be kept up to date with the code. Since changing the stencil requires changing both the analysis and the simulation code, it is difficult to prototype simulations. It also makes it difficult to experiment with different methods of parallelization that might be more effective than the domain decomposition approach since these would require different analyses.

The approach taken in Liszt is to represent the code at a high enough level of abstraction such that the stencil of a piece of Liszt code could be discovered using program analysis. An example of Liszt code is shown in Figure 2.1. To expose the parallelism in the mesh, it provides a `for` statement that operates in parallel over sets of mesh elements. It also builds information about the mesh into the language, using built-in operators to extract local topology such as the vertices on either side of an edge (`head(v)` and `tail(v)`).

Since all data is accessed using these built-in topological operators, it is possible for Liszt to extract the data-dependencies of a piece of code. The Liszt compiler works by extracting this stencil using program analysis and then uses it to automate parallelization via domain-decomposition. Since the stencil analysis is abstract, it can also be reused for other parallelization techniques. For instance Liszt's GPU runtime uses the results of the stencil analysis to divide mesh elements into sets ("colors") that do not have interfering writes, allowing them to be run on the GPU in parallel.

Analysis of stencils in Liszt is dependent on both *static* information (the code being run), and *dynamic* information (the mesh it is being run on). Since Liszt is an offline compiler, the analysis of the stencil needs to be split across both phases. In the compiler, we generate new code that would be a conservative estimate of the actual stencil by executing both sides of divergent control flow as well as relying on language restrictions that limit the stencil to a constant size relative to the code written. At runtime, the code is run across the mesh in a recording mode which discovers the stencil.

The original design of Liszt has major limitations. While it makes it possible to easily prototype the compiler and achieve high-performance, the need for multiple processes to run the compiler and the runtime makes it more difficult to integrate into existing systems.

Unlike autotuners, there is not a single API that can be distributed like a library. In a DSL, users are expected to provide new programs to run. DSLs themselves are never entire programs so they need to be integrated into larger applications. This means that the user of the DSL now needs to know how to run the DSL compiler, and then link the result into their application.

The offline compilation of Liszt strongly couples the design of the Liszt compiler and runtime to the way that it is exposed to other applications. It complicates the stencil analysis transform, forcing us to separate it into a conservative static part and a dynamic analysis. The user of the DSL is then responsible for making sure the results of the compiler were plumbed into the runtime, which runs in a different process. Re-architecting the system to run the stencil-generating analysis dynamically might improve performance and simplify the plumbing for the application developer, but would require compilation during runtime.

2.1.4 *Dynamically-compiled DSLs*

To avoid the limitations of static compilation that complicate deployment, some DSL designs choose to generate code dynamically in the same process. In fact, the most widely adopted DSLs, such as SQL for database processing, and OpenGL/Direct3D for graphics processing are designed as dynamically-compiled languages.

Dynamic compilation simplifies deployment in two ways. First, since there is only one CPU process involved, it can be linked into an existing application like a standard library. This facet is especially important for active libraries, since it makes it possible to use an active library as a drop-in replacement of an offline one. Second, since compilation can occur at any time, it gives the library freedom to change when analyses are performed, potentially specializing the program to dynamically provided information. Structures that would more easily be handled by the compiler in a two-stage design, such as fields in Liszt, can now be handled dynamically, making the DSL more flexible to use in an existing application. Decisions about what code to generate (CPU or GPU, vectorized or scalar, AMD or NVIDIA), can now be made at runtime rather than having to pre-compile all possible combinations.

Dynamic compilation can make deployment easier, but actually compiling code dynamically is more complicated. Approaches such as source-to-source compilation are harder

Written directly in C:

```
float solve(float a, float b, float c) {
  return (-b + sqrt(b*b - 4*a*c)) / 2 * a;
}
```

Programmatically generated using LLVM:

```
Value* float_mul = B.CreateFMul(float_b, float_b);
Value* float_mul1 = B.CreateFMul(float_a, const_float_3);
Value* float_mul2 = B.CreateFMul(float_mul1, float_c);
Value* float_sub3 = B.CreateFSub(float_mul, float_mul2);
Value* float_call = B.CreateCall(func_sqrtf, float_sub3);
Value* float_add = B.CreateFSub(float_call, float_b)
Value* float_div = B.CreateFMul(float_add, const_float_4);
Value* float_mul4 = B.CreateFMul(float_div, float_a);
B.CreateReturn(float_mul4);
```

Figure 2.2: Libraries to generate code such as LLVM allow arbitrary code to be created programmatically, but are often much more verbose than writing the code directly. In this example, the skeleton to set up the surrounding LLVM function that defines variables such as `float_b` is another 30 lines.

to use. In this case, the library must write the source, internally spawn an instance of the compiler to write a dynamic library, and then link the dynamic library back into the program. Since compilation is occurring dynamically, the overhead of using a separate process becomes a concern. Additionally, the generated functions need to be accessed using API calls to the dynamic loader. Dependencies on already loaded code need to be explicitly managed by the DSL, and expressed to the compiler. Dynamic library loading on current operating systems was not designed with frequent JIT compilation in mind, making it necessary for the DSL to deal with issues such as providing globally unique names for all generated functions.

Most dynamically compiling DSLs forgo source-to-source compilation and use another means to JIT compile code. Widely used DSLs such as OpenGL normally use custom compilation frameworks designed by large teams. Others such as the Halide DSL [61] for image processing use library-based compilers such as LLVM [49]. These compilers help manage the details of JIT compiling and linking code. But since they are libraries, actually expressing the generated code is more verbose, as illustrated in Figure 2.2. Because of this verbosity, runtime library code that does not need to be generated dynamically is typically written directly in a low-level language. This split makes it more difficult to move code between runtime and compiler, a task that occurs commonly when prototyping whether domain-specific optimizations are effective.

Furthermore, it is still desirable to express transformations using higher-level languages. Halide, for instance, was originally written in OCaml to make prototyping easier, meaning that three languages were used to write it — OCaml for the transformations, LLVM for generated code, and C/C++ for the high-performance runtime.

2.1.5 *Summary*

A few key points are present across all of the designs used in practice:

1. Almost all techniques in practice target low-level high-performance code at the level of C/C++, or target the assembly directly.
2. Many use high-level languages (OCaml, Scala, Makefiles) to manage their compiler transforms.
3. Every DSL and autotuner does a sophisticated set of transformations across multiple intermediate representations. The original program, or some subset of it, is not simply transliterated into code.
4. Dynamically compiling code can reduce the complexity of deployment and data propagation, but many designs use static compilation since dynamically generating low-level code efficiently requires verbose libraries such as LLVM.

2.2 Existing techniques for code generation

To support how programmers design DSLs and active libraries, a tool should make it possible to (1) use a low-level language for generated code, (2) express compiler transforms in a high-level language, (3) substantially transform input code, and (4) perform code generation dynamically. Currently existing language-based approaches for code generation in the literature accomplish some but not all of these goals.

2.2.1 *LISP*

One of the oldest designs for generating code dynamically is found in the LISP [52] language and its derivatives. In LISP, both code *and* data are expressed as nested lists of elements. The syntax of the language makes these lists explicit using s-expressions based on parentheses. Since the syntax closely resembles the data-structure of the code itself, it is easy to reason about code that generates code. Code is just another form of data — everything is "just lists." A language like LISP whose syntax for code resembles its data-structures is said to be *homeoiconic*.

LISP programs use homeoiconicity to make code generation easier. Standard list manipulation functions can construct code, and any list can be used as code by evaluating it using the `eval` form. It is easy to inspect the lists that represent code, and their simple syntax means that novice users can understand how generated code may evaluate. By default a LISP interpreter runs `eval` on the file being executed. It may also run a macro processor beforehand. This macro processor looks for special constructs that define *macros*, functions that operate on unevaluated lists before evaluation. It then expands these macros by running the code. Many constructs in LISP distributions are implemented as *macros* rather than built-ins using this mechanism.

The LISP model has been used in the past to create DSLs. In particular, Racket [75] provides an interface to the static semantics of the language using macros. Using this interface they implement a typed variant of Racket, as well as other DSLs. The macro system is used to translate typed Racket to standard Racket with a few extensions to support unchecked access to fields.

LISP's simple model for code generation is powerful, but it is not ideal for generating high-performance code since the language itself is very high-level. In the general case, it is dynamically typed and garbage collected. Additionally, since LISP can generate more LISP code, it is possible that the *generated* LISP code might decide to create and evaluate more code. Even when implemented well, these features may introduce unpredictable performance overheads, making it hard to reason about lower-level details such as cache behavior.

```
; A macro that uses an internal temporary variable 'temp':
> (defmacro addone (x)
    `(let ((temp 1))
       (+ ,x temp)))
ADDONE

; The macro appears to work fine for most expressions
> (let ((a 3)) (addone a))
4
; But it fails when the temporary name shadows variable declarations used in the input:
> (let ((temp 3)) (addone temp))
2

; We can fix this problem by generating a unique name for the temporary with gensym
> (defmacro addone (x)
    (let ((temp (gensym)))
      `(let ((,temp 1))
         (+ ,x ,temp)))
```

Figure 2.3: Simple meta-programming systems such as that found in Common LISP can lead to surprising errors when a variable introduced in the implementation of a macro shadows a variable used in an argument to the macro. In the above example, the `temp` variable definition in the `addone` macro shadows the `temp` variable reference in the calling code. These macro systems are called unhygienic since arbitrary naming choices in the macro's implementation can change the semantics of the caller's code. In these systems, correctly implemented macros use uniquely generated names (using the `gensym` function) to avoid accidental capture.

Some LISP derivatives [75, 65] have added escape hatches to these behaviors. For instance Common LISP allows a programmer to forgo dynamic type checks with the right annotations. But even if one programmer writes code without these high-level features, libraries may still rely on them, making it hard to write large amounts of code without them. Furthermore, features like dynamic typing, garbage collection, and code generation require more effort to port to architectures such as GPUs and small embedded CPUs.

Another issue arises with homeoiconic forms. When variables are represented as simple identifiers, it is easy to generate code where the value of a variable is accidentally captured by a variable definition of the same name introduced by generated code. Figure 2.3 shows an example in LISP. Ideally, visual inspection of the code should make the relationship between defined and used variables clear. So some LISP derivatives, Scheme in particular, define macro expansion in a way that is *hygienic* [47]. An API for dealing with identifiers uses the special forms `syntax-rules` and `syntax-case` to match on identifiers and generate code with them that avoids accidental capture.

Macro hygiene solves the variable capture problem, but to generate code, the programmer needs to know how to use an API for identifiers. This API is now one level removed from the simple homeoiconic forms, reducing the benefit of having a simple data structure for code in the first place.

2.2.2 *Partial evaluation*

Rather than write code generators explicitly, another approach to creating high-performance code is to *specialize* generic programs using an automated process. Approaches based on *partial evaluation* make some decision about which inputs to a generic function should be fixed to a particular value and then attempt to optimize the code by pre-computing any values possible ahead of time without modifying the semantics of the program [44, 43]. In this model, optimizing an operation like a matrix-multiply, or an FFT might work by writing a generic version of the algorithm that works over different block sizes or problem sizes, followed by an automated specialization that chooses to make the block size or problem size constant.

Partial evaluation has some advantages over direct code generation for certain tasks. Its model for directing optimization is very lightweight and high-level. The user only needs to specify what values should be specialized and what should remain variable at runtime. It may be simpler to reason about the semantics of a program if it contains no explicit code generation. If input is type-safe, then we are guaranteed that the specialized code generated via partial evaluation is as well, something that is difficult to guarantee if generating arbitrary code directly.

The partial evaluator can also exist as a separate pass from the language itself. The language being specialized does not need any support for manipulating code through data structures like lists in LISP. This allows partial specializers to work on low-level languages where writing code generators would be tedious. In fact, many partial specialization frameworks have been developed for C [18, 35].

Just like code generation in LISP, the process of partial specialization can be made to run dynamically as well. For instance DyC [35], a partial specializer for C, allows specialization to occur at runtime and uses dynamic compilation of templates to optimize performance.

It may seem like some forms of code generation such as generic compilers would not be expressible using partial evaluation. But a common approach is to use partial evaluation to turn an *interpreter* into a *compiler* [31]. The interpreter reads an intermediate representation of the program and evaluates the behavior described in the program. Partial evaluation is then used to specialize on the particular input program, which effectively creates a compiler out of an interpreter.

The disadvantage of partial evaluation is that it is not always easy to control, or even understand, what optimizations will be done. An offline partial evaluation framework includes some form of *binding-time analysis*, which marks which expressions are constant and which are runtime dependent given some constant input values [17]. This analysis must be conservative. The partial evaluator will not know if some expression will terminate and typically must bound evaluation to ensure that the partial evaluator itself will terminate. The process is also frequently path insensitive so conservative decisions are made about what can be evaluated when different branches of control meet. The limitations about what can be evaluated are analogous to those found in traditional compiler optimization passes such as constant propagation and common subexpression elimination. Even in online partial evaluation, where all specialization is done only when the input values are known, it might still be worthwhile to speculatively execute expressions that are only evaluated under some paths if they would reduce to simpler computations.

A compiler that is generated from an interpreter via partial evaluation may suffer from these limitations. Compiling an interpreter loop will inline the functions that run individual operations on each IR node. These operations in the interpreter might refer to a virtual stack, register set, or other data-structure used to implement the interpreter. In addition to simply evaluating the interpreter loop, a good partial evaluator needs to be able to remove these data-structures as well, promoting accesses from the virtual stack into actual machine registers for local values. In this case, the virtual stack is only partially known. Some of it will become the actual function-call stack for the compiled program, which is created dynamically. In these cases it can be difficult to perform accurate alias analysis to know the dynamic and static parts of a data-structure. The result is that a programmer does not know exactly what the code would look like after compilation.

The uncertainties that result from performing partial evaluation on programs are not appropriate for high-performance examples like those we examined in the previous section, since they often rely on precise layout of data and execution order to achieve good memory performance. Furthermore, removing the need to do explicit code generation becomes less useful when programs already have to do substantial analysis and transformations of code to get high performance. The DSLs we described in Section 2.1 require major transformations before code emission. Regardless of the technique to express generated code, the majority of the system will still require the same effort in reasoning about code transformations as a normal compiler during its optimization. Once most of the system is compiler-like in its functionality, it becomes awkward to use partial evaluation just to generate the code.

2.2.3 *Multi-stage programming*

Part of the motivation for multi-stage programming was improving the lack of control in partial evaluation systems [73]. Rather than allow the entire program to be partially evaluated based on a few directives about what is static and dynamic, multi-stage programming uses specific annotations that indicate when an expression should be evaluated. These annotations can be marked at the level of *expressions* or at the level of *types*.

MetaML [73] and Meta-OCaml [72] provide expression-based annotations for the ML family of languages. A quote operator constructs an unevaluated fragment of ML code, while an escape operator introduces a hole in that fragment that will be filled in by evaluating the code in the escape at an earlier stage. Figure 2.4 shows an example of the operators in practice. In this way, multi-stage programming is similar to constructing code in LISP. In LISP, code was created by manipulating lists using standard operators. In multi-stage programming, code is constructed and manipulated using built-in operators.

In multi-stage programming, staging annotations are viewed as a way to control the optimization of code. Since high-performance code is often found through experimentation, there is a desire to make staging decisions more modular so different ways of staging code can be discovered. Deciding whether to unroll a loop is one example of such a choice. In expression-based staging, the choice whether to unroll a loop is encoded in the loop expression explicitly. A different instance of the loop needs to be written for the unrolled case. To make the code more modular, there has been work on staging that is directed

```
(* a quote operator < constructs a fragment of OCaml code. This form is
analogous to creating a string literal, except we are constructing code
rather than strings. In this case the code when run will produce an
integer so it has type ‘‘code of int’’ or <int> *)
val code = <1 + 2>;

(* an escape operator ˜ splices one fragment of OCaml code into another.
It is analogous to string interpolation, where another fragment of code
is inserted into a template *)
val code2 = <3 + ˜code>

(* These can be combined together to write functions that generate code.
Here we show an example that generates specialized power functions, bˆe,
for a particular exponent *)
fun exp (b, e) =
  if e = 0 then <1.0>
         else <b * ˜exp(b,e-1) >

(* run executes the staged code fragment, lowering into the current
stage. Running a function is analogous to compiling the code *)
val pow3 = run <fun x -> exp(b,3)>
```

Figure 2.4: Multi-stage programming in MetaOCaml. Operators create quotations of code, and escapes allow the programmer to stick code together through interpolation.

by annotations on *types* rather than expressions, such as light-weight modular staging in Scala [62]. In these frameworks, the *type* of the collection being looped over determines whether it should be unrolled or not. The programmer can then expose the choice of whether or not to unroll the loop as a type parameter.

In both the type-directed and expression-directed cases, staged programming focuses on accelerating existing programs by adding annotations and making it as easy as possible to change staging decisions. These approaches have been applied to improve the performance of code. For instance, Carette investigates staging of Gaussian elimination in MetaOCaml [10], while Cohen et al. investigate applying MetaOCaml to problems in high-performance computing like loop unrolling and pipelining [16].

For the purposes of creating high-performance DSLs and active libraries this focus has some limitations. The desire to switch what is staged and what is not staged encourages the staged language and the staging language to be the same (*homogeneous* staged programming). Attempts at low-level staged programming such as the ‘C language [58] can generate low-level code, but also require the generators for that code to be written at a low level as well.

The DSLs we examined in Section 2.1, in contrast, explicitly used different languages to write program transformations and exert control over performance. A small amount of work has looked at *heterogeneous* multi-stage programming to address this problem. MetaOCaml can use an approach called *offshoring* where a subset of the original language can be cross-compiled to a lower-level C language. For instance, Eckhardt et al. propose implicit heterogeneous programming in OCaml with a translation into C [25]. This approach limits what can be in the low-level language to a subset of the original language, and it may be difficult to understand what is contained in that subset. For instance in the system of Eckhardt et al. the type language is limited to primitive types and arrays. Other heterogeneous systems exist as well. For instance, MetaHaskell is an extension of Haskell for heterogeneous meta-programming that supports embedding new object languages [51].

Another limitation arises from the focus on type-correctness. Like partial evaluation, most multi-stage programming frameworks, include MetaML, Meta-OCaml, and Meta-Haskell statically ensure the program is type correct. In the case of explicit staging annotations, this correctness both means that normal code is type-safe, and that any *generated* code that might possibly be created during staging is also type-safe. Since types need to be checked statically, it makes it more difficult for these languages to generate new *types* using staging. Nevertheless, we have found that generated types can make it easy to create concise active libraries.

2.3 Conclusion

In this chapter, we have seen that practical DSL and active library development uses high-level languages to generate high-performance low-level code. These libraries perform sophisticated transformations and analysis of intermediates, and their designs are simplified with dynamic compilation. Each of the principled approaches to code generation—LISP, partial evaluation, and multi-stage programming—supports only some of these design patterns. The LISP approach provides a simple model for generating code, but only allows the generation of high-level code limiting performance. Furthermore, the model that all code is simply lists of symbols lacks hygiene and adding hygiene takes away some of its simplicity. Partial evaluation approaches like DyC [35] make it possible to express optimizations even in low-level languages such as C, but actually controlling what code gets generated can be difficult. It is hard to know what quality of code will result after partially evaluating an interpreter into a compiler.

Multi-stage programming improves on the control of optimization that partial evaluation lacks. It also has simple operators, and since those operators work at the level of code fragments rather than simple lists, it is able to preserve lexical scoping by default. So far, however, staged programming has primarily been done purely in high-level languages or low-level languages. For most work, the focus has been type-correctness of staged code rather than raw performance of generated code.

The design that we propose for Terra extends the techniques in previous work so that they follow the way programmers create current designs in practice, which leads to a different design from previous work. We explicitly use two separate languages for the compiler and generated code. We use multi-stage programming operators to ensure the programmer has control of what code is produced. And we architect the system so that code generation can be performed dynamically to reduce the complexity of deploying code generators.

CHAPTER 3
GENERATING CODE

As we saw in the previous chapter, practical DSLs and autotuners used high-level languages to generate high-performance low-level code. We support this approach by explicitly using a *two-language* design. A *high*-level language, Lua, is used to write transformations. The features of Lua makes it easy to manage the process of code generation. Lua has automatic memory management, simplifying the process of storing intermediate representations which often take the form of large self-referential graphs. Higher-order functions can be used to optimize intermediate representations or represent other transformations.

Terra itself is a *low*-level language that serves as the target for code generation. The low-level nature of Terra makes it possible to achieve high performance. It includes ways to explicitly layout memory, and use hardware vector instructions to maximize compute throughput. The approach of supplementing dynamically-typed languages with a high-performance language has been used successfully before. For instance, Cython is an extension to the Python language that allows the creation of C extensions while writing in Python's syntax [5] and Copperhead supplements Python with a vector-style language that can run on GPUs [11]. But these approaches only allow the programmer to write high-performance code directly ahead-of-time, rather than dynamically generating code at runtime.

We support the generation of code at runtime by incorporating multi-stage programming operators in Lua that allow it to stage Terra code in a principled way. The staged programming of Terra provides interoperability between compiler, generated code, and runtime of a DSL. We will see that many higher-level language features, such as namespaces, templating, or autotuning simply arise from using Lua as the environment for creating Terra code.

In contrast to partial evaluation, the explicit staging operators make it clear what Terra code is being produced, giving clear control over performance. Unlike most multi-stage programming techniques, we focus on making explicit generation of next-stage code easier, rather than making it possible to have modular annotations. This is because we believe most existing applications that get high-performance will need significant transformation even before code generation, making the modularity of annotations less important.

We also take a unique approach to typechecking. In LISP all code is dynamically-typed. In multi-stage programming, all code, generated and static, is statically-typed. For Terra, the top-level code is dynamically typed like LISP because Lua itself is a dynamically-typed language. This dynamic typing gives the programmer the freedom to generate whatever code and types are desired at runtime. Then, multi-stage programming operators are used to generate the low-level Terra language. Terra does contain types, which will be checked when Terra code is *compiled* (during Lua evaluation). This design allows the generation of low-level code, and provides type errors before that code is run.

Finally, despite being different languages, we run both Lua and Terra in the same process, allowing each language to call functions in the other.

We introduce Terra and Lua by example, starting with simple functions. We then show how the multi-stage programming operators work in Lua and Terra, and use this infrastructure to show how we can build and optimize a simple DSL for image processing. We then discuss design decisions that enable the concise generation of low-level code for building DSLs. A formal definition of the semantics for a simplified version of the languages is presented in the next chapter.

3.1 Writing code in Terra

At the top level, a program evaluates as Lua code. For instance, a programmer may define a function and print the result:

```
function min(a, b)
  if a < b then return a
  else return b end
end
print(min(3,4)) -- 3
```

We augment the normal set of Lua statements and expressions with special constructs that create Terra functions, types, variables, and expressions. The `terra` keyword introduces a new Terra function and can appear where a Lua expression appears:

```
terra min(a: int, b: int) : int
  if a < b then return a
  else return b end
end
print(min(3,4)) -- calling Terra function from Lua
```

Terra functions are lexically-scoped and statically-typed, with parameters and return types explicitly annotated. Terra functions can also be called directly from Lua.

Terra entities (functions, types, variables and expressions) are first-class Lua values. So, for example, after we define the `min` Terra function, the *Lua* variable `min`'s value is a Terra function definition. That definition can be used like any other value in Lua. It can be passed and returned from functions or stored in Lua data structures.

Terra is also backwards-compatible with C, a feature we will use in our DSLs and is discussed further in Chapter 10:

```
std = terralib.includec("stdlib.h")
terra main()
    std.printf("hello, world\n")
end
main()
```

The Lua function `includec` imports the C functions from `stdlib.h`.

The programmer can use the fact that Terra entities are first-class to organize Terra code. The function `terralib.includec` is itself a good example of this behavior. It creates a Lua *table*, an associative map. It then fills the table with Terra functions that invoke the corresponding C functions found in `stdlib.h`. In Lua, the expression `table.key` is syntax sugar for `table["key"]`. In the above example, the call to `std.printf` will resolve to C's `printf` and since `std` is a Lua table, Terra will resolve the lookup during the compilation process, avoiding the runtime overhead for the table lookup.

To describe user-defined Terra types, we also introduce a construct `struct`. Here we can use it to hold a square greyscale image:

```
struct GreyscaleImage {
  data : &float;
  N : int;
}
```

`GreyscaleImage` is a Lua variable whose value is a Terra type. Terra's types are similar to C's. They include standard base types, arrays, pointers, and nominally-typed structs. Here `data` is a pointer to `float`s, while `GreyscaleImage` is a type that was created by the `struct` constructor.

We might want to parameterize the image type based on the type stored at each pixel (e.g., an RGB triplet, or a greyscale value). Because all Terra entities are first-class, we can define a Lua function `Image` that creates the desired Terra type at runtime. This is conceptually similar to a C++ template:

```
  function Image(PixelType)
    struct ImageImpl {
      data : &PixelType,
      N : int
5   }
    -- method definitions for the image:
    terra ImageImpl:init(N: int): {} --returns nothing
      self.data =
        [&PixelType](std.malloc(N*N*sizeof(PixelType)))
10    self.N = N
    end
    terra ImageImpl:get(x: int, y: int) : PixelType
      return self.data[x*self.N + y]
    end
15  --omitted methods for: set, save, load, free
    return ImageImpl
  end
```

In addition to its layout declared on lines 2–5, each `struct` can have a set of methods (lines 6–15). Methods are normal Terra functions stored in a Lua table associated with each type (`ImageImpl.methods`). The method declaration syntax is sugar for:

```
ImageImpl.methods.init =
  terra(self : &ImageImpl, N : int) : {}
    ...
  end
```

Method invocations (`myimage:init(128)`) are also just syntactic sugar:

```
ImageImpl.methods.init(myimage,128)}
```

In Chapter 6, we show how we also allow user-defined behavior in this de-sugaring process to define custom method invocation semantics.

In the `init` function, we call `std.malloc` to allocate memory for our image. We also define a `get` function to retrieve each pixel, as well as some utility functions which we omit for brevity.

Outside of the `Image` function, we call `Image(float)` to define `GreyscaleImage`. We use it to define a `laplace` function and a driver function `runlaplace` that will run it on an image loaded from disk to calculate the Laplacian of the image:

```
GreyscaleImage = Image(float)
terra laplace(img: &GreyscaleImage,
              out: &GreyscaleImage) : {}
  --shrink result, do not calculate boundaries
  var newN = img.N - 2
  out:init(newN)
  for i = 0,newN do
    for j = 0,newN do
      var v = img:get(i+0,j+1) + img:get(i+2,j+1)
            + img:get(i+1,j+2) + img:get(i+1,j+0)
            - 4 * img:get(i+1,j+1)
      out:set(i,j,v)
    end
  end
end
terra runlaplace(input: rawstring,
                 output: rawstring) : {}
    var i = GreyscaleImage {}
    var o = GreyscaleImage {}
    i:load(input)
    laplace(&i,&o)
    o:save(output)
    i:free(); o:free()
end
```

To actually execute this Terra function, we can call it from Lua:

```
runlaplace("myinput.bmp","myoutput.bmp")
```

Invoking the function from Lua will cause the `runlaplace` function to be JIT compiled. A foreign function interface converts the Lua string type into a raw character array `rawstring` used in Terra code.

More generally, we treat compilation of Terra functions as an explicit operation in the language. The same Terra function can be compiled for different purposes. For offline use, we can save the Terra function to a shared library (`.so` file) which can be linked to another process using the built-in `saveobj` function:

```
terralib.saveobj("runlaplace.so",
                 {runlaplace = runlaplace})
```

The second argument is a Lua table that contains the Terra functions we want to compile and export. Another built-in function, `cudacompile`, takes Terra functions and compiles them into CUDA kernels that can run on GPUs. In the future we imagine extending the interface for explicit compilation to support additional architectures.

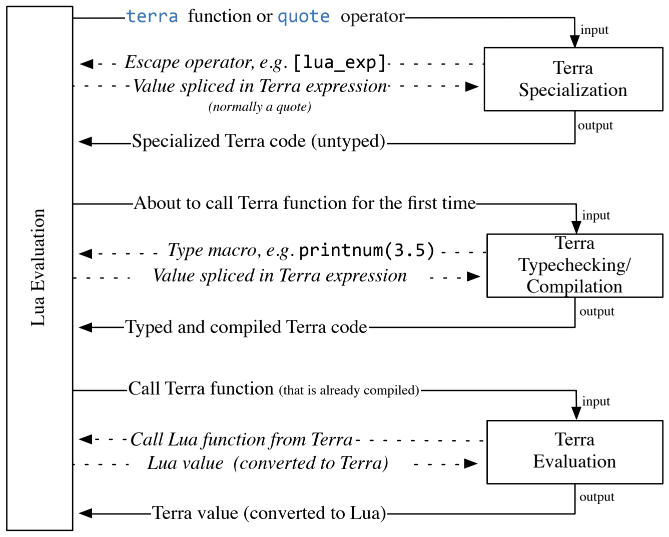

Lua Evaluation
terra function or quote operator
input
Escape operator, e.g. [lua_exp]
Value spliced in Terra expression
(normally a quote)
Terra Specialization
output
Specialized Terra code (untyped)
About to call Terra function for the first time
input
Type macro, e.g. printnum(3.5)
Value spliced in Terra expression
Terra Typechecking/ Compilation
output
Typed and compiled Terra code
Call Terra function (that is already compiled)
input
Call Lua function from Terra
Lua value (converted to Terra)
Terra Evaluation
output
Terra value (converted to Lua)

Figure 3.1: Phases of evaluation of a Lua-Terra program.

3.2 MULTI-STAGE PROGRAMMING IN TERRA

In addition to making Terra entities first class, Terra includes explicit multi-stage programming operators like those found in MetaML [73] and MetaOCaml [72]. In those languages the staged programs are typechecked statically. In contrast, Terra code is typechecked at runtime. To support this behavior, staged programming in Lua and Terra involves *two phases* of meta-programming, illustrated in Figure 3.1. First, untyped Terra expressions are constructed using *quotations* and stitched together using *escapes* in a process we call *specialization*.[1] This process is similar to what occurs in other multi-stage languages. Second, type-level computation can be carried out during typechecking with user-defined *type-macros*, which allow the programmer to generate code based on the *type* of an expression. This phase does not exist in other staged programming frameworks since the types must already be determined statically. Here we focus on generating code, but in Chapter 6, we will show how this second stage can also be used to flexibly generate high-performance types. The interaction between these phases of meta-programming and the evaluation of Lua and Terra code is summarized in Figure 6.1.

During specialization, a quotation (the backtick operator `` `exp ``, or the block structured `quote <exps> end`) used in Lua code creates an unevaluated Terra expression, and an escape

[1]So named because it is analogous to the specialization phase in partial evaluation frameworks.

(the bracket operator `[lua_exp]`) used in Terra code evaluates `lua_exp` and splices its result (normally a Terra quotation) into the surrounding Terra code.

To understand how the phases of execution interact, we can consider how to use these operators to generate a specialized version of `powf` for a particular value of `N`:

```
   function genpowf(N)
       local function genexp(vr)
           local r = `1.0
           for i = 1,N do r = `([r] * [vr]) end
5          return r
       end

       local terra powfN(v : double)
           return [genexp(`v)]
10     end

       return powfN
   end
   pow2 = genpowf(2)
15 print(pow2(3)) -- '9'
```

We begin by evaluating Lua expressions, invoking `genpowf(2)`, which defines `genexp` and then defines the Terra function `powfN`. When a Terra function or quotation is defined, it is specialized in the local environment. Specialization resolves the escaped Lua expressions by calling back into Lua evaluation, splicing the resulting values into the Terra code. In `powfN`, it evaluates the escaped call to `genexp`, which will generate the body of `powfN`. The loop on line 4 alternates between defining a Terra quotation `` `([r] * [v]) ``, and specializing it with values of the Lua variables `r` and `vr`. Here `r` holds the power expression being built `` `1.0*v*... ``, while `vr` is a quotation of a variable that refers to parameter `v` of `powfN`. The result of the loop is the Terra quotation `` `1.0 * v * v ``, which will be spliced into the body of `powfN`, completing its specialization.

When a Terra function is first called, such as `pow2` on line 13, it is *typechecked and compiled*, producing machine code. The function is then evaluated computing the result `9`.

The `pow2` function does not require any meta-programming during the typechecking process. However, in some cases, it is useful for the generating code to be aware of the type of an expression. The *type macro* allows for user-defined behavior during typechecking. These macros can be used in Terra code like a function, but they are evaluated when the Terra code is typechecked. Unlike escapes and quotes, which are used to create and stitch code together before typechecking, type-macros have access to their arguments' types. They can be used to generate behavior based on the types as shown in this example:

```
   printnum = macro(function(num)
       local format
       if num:gettype() == float then
         format = "%f"
5      else
         format = "%d"
       end
       return `C.printf([format],[num])
   end)
10 terra printint(a : int) printnum(a) end
   printint(1)
```

When `printint` is first called on line 11, it will be typechecked. When typechecking the call `printnum(a)`, the typechecker will invoke the `printnum` macro, which examines the type of the argument `num` to generate the appropriate formatting code for the type.

3.3 Writing optimizations using staging

We can also use multi-stage programming to implement higher-order Lua functions that do performance optimizations. For instance, we may want to optimize the `laplace` function from Section 3.1 by blocking the loop nests to make the memory accesses more friendly to cache. We could write this optimization manually, but the sizes and numbers of levels of cache can vary across machines, so maintaining a multi-level blocked loop can be tedious. Instead, we can create a Lua function, `blockedloop`, to *generate* the Terra code for the loop nests with a parameterizable number of block sizes. In `laplace`, we can replace the loop nests (lines 7–12) with a call to `blockedloop` that generates Terra code for a 2-level blocking scheme with outer blocks of size 128 and inner blocks of size 64:

```
[blockedloop(newN,{128,64,1}, function(i,j)
  return quote
    var v = img:get(i+0,j+1) + img:get(i+2,j+1)
          + img:get(i+1,j+2) + img:get(i+1,j+0)
          - 4 * img:get(i+1,j+1)
    out:set(i,j,v)
  end
end)]
```

The escape (`[]`) around the expression allows the loop nest generated by `blockedloop` to be spliced into the Terra expression. The function `blockedloop` itself is a higher-order Lua function whose third argument is a Lua function that is called to create the inner body of the loop. Its arguments (`i`,`j`) are the loop indices. The `quote` expression defines the loop body using the loop indices. The implementation of `blockedloop` walks through the list of `blocksizes`:

```
function blockedloop(N,blocksizes,bodyfn)
  local function generatelevel(n,ii,jj,bb)
    if n > #blocksizes then
      return bodyfn(ii,jj)
    end
    local blocksize = blocksizes[n]
    return quote
      for i = ii,min(ii+bb,N),blocksize do
        for j = jj,min(jj+bb,N),blocksize do
          [ generatelevel(n+1,i,j,blocksize) ]
        end
      end
    end
  end
  return generatelevel(1,0,0,N)
end
```

It uses a quote to create a level of loop nests for each entry and recursively creates the next level using an escape.At the inner-most level, it calls `bodyfn` to generate the loop body. A more general version of this function is used to implement multi-level blocking for our matrix multiply example.

This example highlights some important features of Terra that we have presented so far. We provide syntax sugar for common patterns in runtime code such as namespaces (`std.malloc`) or method invocation (`out:init(newN)`). Furthermore, during the generation of Terra functions, both Lua and Terra share the same lexical environment. For example, the loop nests refer to `blocksize`, a Lua number, while the Lua code that calls `generatelevel` refers to `i` and `j`, Terra variables. Values from Lua such as `blocksize` will be specialized in the staged code as constants, while Terra variables that appear in Lua code such as `i` will behave as variable references once placed in a Terra quotation.

3.4 Generating arbitrary code using staging.

Staging operations allow the generation of arbitrary code, and we can use them to generate high-performance code for DSLs. In Section 3.1, we showed a piece of image processing code written as an explicit loop. We could instead represent this operation using a small DSL. Image processing DSLs such as Halide [61] express image processing operations as image-wide operations that can be composed. For our example, we might express the Laplace operator similarly:

```
laplace = img(x-1,y) + img(x+1,y) + img(x,y-1) + img(x,y+1) -  4 * img(x,y)
```

This way of expressing the problem is advantageous because it separates the algorithm from optimizations such as threading, blocking, or loop vectorization. It can speed up image processing code by an order of magnitude [61]. A frontend to a DSL would typically parse this code into an intermediate representation. Chapter 9 discusses Terra's approach to DSL frontends, but for now we can imagine that the frontend has converted the text into an intermediate representation like the following, which represents the same Laplace operator as before using Lua tables:

```
local laplace =
{ "-",  { "+",  {"+", {"load",-1,0},
                      {"load",1,0}},
                {"+", {"load",0,-1},
                      {"load",0,1}}},
        {"*", {"const",4},
              {"load", 0, 0 }}}
```

For simplicity, we use a LISP-like style, where the first element in a Lua table indicates the operator (e.g., `"+"`, or `"load"`) and arguments to that operator follow. To compile this IR into high-performance Terra code, we can create a Lua function, `compileimage`, which takes the IR as an argument and returns a Terra function that actually performs the operation given an input image.

One possible implementation of this function in shown in Figure 3.2. It returns a Terra function that implements the skeleton of the image processing loop, initializing the output image and then looping over the pixels. The inner loop, however, is dependent on the IR itself, so we use an escape operator to call the function `expr` which is actually responsible for generating the computation from the IR. It matches on the type of the IR node (e.g. `"+"`, or `"load"`), and then produces a Terra quotation that implements the particular operator. Some IR nodes such as `"+"`, contain further expressions, so they call the `expr` function recursively. We also use a local Terra function, `load`, to handle loading data from an image with the correct behavior for the boundaries, returning 0 for any access outside the input image. This illustrates how easy it is for the generated code to simply call into library functions like `load`.

```
function compileimage(ir)
  -- load image with black boundary conditions
  local terra load(img : &GreyscaleImage,i : int,j : int)
    if i < 0 or j < 0 or i >= img.N or j >= img.N then
        return 0
    else
        return img:get(i,j)
    end
  end
  local function expr(node,i,j,input)
    local op,e0,e1 = node[1],node[2],node[3]
    if "const" == op then
        return `e0
    elseif "load" == op then
        return `load(input,i+e0,j + e1)
    elseif "+" == op then ...
  end
  return terra(input : &GreyscaleImage,
               output: &GreyscaleImage) : {}
      output:init(input.N)
      for i = 0,output.N do
        for j = 0,output.N do
          var v = [ expr(ir,i,j,input) ]
          output:set(i,j,v)
        end
      end
    end
end
```

Figure 3.2: An example function that compiles a simple IR for image processing code into a Terra function

Real DSLs will have larger runtime libraries, so making it easy for the generated code to access them is important.

Now that we have a simple code generator for our imaging DSL, the DSL designer might consider ways of optimizing the input. For instance, we could vectorize the inner loop by running multiple pixels at a time. We can use Terra's built-in vectors types to accomplish this, changing the loop over `j` to step a vector at a time, and then changing the implementation of `load` to load a vector of pixels rather than a single one. We could also perform loop blocking using the higher-order `blockedloop` function presented earlier.

We can automatically tune the algorithm with another simple function. It is not clear how long the vectors should be, or what the block size should be for a particular program. So we can modify the `compileimage` function to take the block and vector sizes as parameters, and then try different options. An example of this approach is shown in Figure 3.3.

```
local potentialvectorsizes = {2,4,8,16,32,64}
local potentialblocksizes = {32,64,128,256}
local function tuneimage(ir, inputimage, outputimage)
    local besttiming,bestcandidate = math.huge,nil
    for blocksize in ipairs(potentialblocksizes) do
        for vectorsize in ipairs(potentialvectorsizes) do
            local candidate = compileimage(ir,blocksize,vectorsize)
            local timing = timerun(function()
                candidate(inputimage,outputimage)
            end)
            if timing < besttiming then
                besttiming,bestcandidate = timing,candidate
            end
        end
    end
    return bestcandidate
end
```

Figure 3.3: A simple autotuner for the image processing language.

Actual DSLs and auto-tuners written in Terra take the same form as these examples. They are simply compositions of built-in operators for parameterizing, generating, and running Terra code.

3.5 Design decisions for code generation

In the previous section, we were able to use Lua and Terra to create a concise DSL using just a few operators to generate, run, and auto-tune Terra code. The ability to create these types of programs relies on a few design decisions we made to create a two-language meta-programming system.

3.5.1 *Shared lexical environment*

During specialization of Terra code, Lua and Terra share the same lexical environment. That is, variables that are lexically in scope in Lua context can be used directly inside of Terra code, and variables introduced in Terra code are in scope in escape operators. Other heterogeneous staged programming languages, such as MetaHaskell [51], do not have this shared lexical scope, making it necessary for quotations in one language to always escape any references to the surrounding environment. This design can make even simple programs complex to read. Without shared lexical scope, for instance, the code generator for image processing from the previous section would need many different escapes:

```
-- with shared scope by default:
return `load(input,i + e0,j + e1)

-- without:
return `[load]([input],[i]+[e0],[j]+[e1])
```

While some heterogeneous multi-stage languages with shared lexical scope have occurred organically in the past [25, 76], these are achieved through *offshoring*—reinterpreting the embedding language syntax using a different lower-level semantics, which limits what can be expressed in the lower-level language. In contrast, we support a shared lexical environment across two fully distinct languages.

The shared environment allows us to remove a surprising amount of functionality from Terra itself that a standalone language would require. For instance, Terra itself has no syntax for referring to other Terra functions or any global values at all! That functionality is handled through specialization, where identifiers in Terra code are resolved to the Lua objects that represent the functions or globals. Terra functions can be organized into tables in the Lua environment, and shared scoping allows us to refer to them directly from Terra code without explicit escape expressions. Specifically, we treat any reference to a variable `x`, as if it were escaped by default `[x]`. The details of how this is accomplished are reflected in the semantics presented in Chapter 4. We also treat table selection operators $\texttt{x.id}_1\texttt{.id}_2\texttt{...id}_n$ (where $\texttt{id}_1\texttt{...id}_n$ are valid entries in nested Lua tables) as if they were escaped. This syntactic sugar allows Terra code to refer to functions organized into Lua tables (e.g., `std.printf`), removing the need for an explicit namespace mechanism in Terra.

3.5.2 *Hygienic code generation*

When performing code generation, the programmer needs to understand the relationship between variable definitions and their uses. As seen in some varieties of LISP, even simple macros can unexpectedly capture the values of user-provided variables if not written correctly.

Terra ensures that code generation using Terra variables is hygienic by default. Variable references are lexically scoped:

```
local addone = macro(function(x)
    return quote
        var y = 1 -- variable 'y' covers user-provided expression 'x'
    in
        x + y
    end
end)
terra test()
    var y = 2 -- But this 'y' is unique from the 'y' in the macro
    return addone(y) -- 3
end
```

Maintaining hygiene during staging ensures that it is always possible to determine the relationship between variables and their declarations (across both Lua and Terra) using only the local lexical scope.

3.5.3 *Separate evaluation of Terra code*

After Terra code is compiled, it can run independently from Lua and does not interact with its environment.[2] This is a departure from some other approaches in multi-stage programming. For instance, in MetaOCaml [72], `ref` cells share the same store across different stages, allowing mutations in staged code to be seen outside of the staged code. This alternative makes sharing state easier, but it would couple Terra and Lua's runtimes, which can limit the ability to reason about Terra code independently. The approach of separating compile-time and runtime-time environments has be used for LISP macro expansion [27] to make components more composable. Here we apply the same technique to ensure predictable performance.

In our case, the reliance on the Lua runtime, which includes high-level features such as garbage collection, would make it more difficult to reason about the performance of Terra code. Furthermore, the required runtime support would make it difficult to compile Terra functions on architectures such as GPUs, run code in multiple threads, or link code into existing C programs without including the Lua runtime. Currently these behaviors are all possible because we evaluate Terra code separately.

[2]We do allow for this behavior when specifically requested, but do not provide it by default.

3.5.4 *Eager specialization with lazy typechecking*

Statically-typed languages such as Terra are normally compiled ahead-of-time, resolving symbols, typechecking, and linking in a separate process from execution. Many staged programming languages such as MetaOCaml maintain this static typechecking even for staged code. However, since Lua is dynamically-typed and can generate arbitrary Terra code, it is not possible to typecheck a combined Lua-Terra program statically. Instead, the normal phases of Terra compilation become part of the evaluation of the Lua program, and we must decide when those phases run in relation to the Lua program. To better understand how Terra code is compiled in relation to Lua, consider where Terra can "go wrong." While *specializing* Terra code, we might encounter an undefined variable, resolve a Lua expression used in an escape to a value that is not also a Terra term expression, or resolve a Lua expression used as a Terra type to a value that is not a Terra type. While *typechecking* Terra code, we might encounter a type error. And, while *linking* Terra code, we might find that a Terra function refers to a declared but undefined function. In Terra, we perform specialization *eagerly* (as soon as a Terra function or quotation is defined), while we perform typechecking and linking *lazily* (only when a function is called, or is referred to by another function being called).

Eager specialization prevents mutations in Lua code from changing the meaning of a Terra function between when it is defined and when it is used. Eager specialization requires all symbols used in a function to be defined before it is used, which can be problematic for mutually recursive functions. In order to support recursive functions with eager specialization, we allow the separate declaration and definition of Terra functions:

```
terra isodd -- function declarations

terra iseven(n : int) : bool
    if n == 0 then return true
    else return isodd(n - 1) end
end

terra isodd(n : int) : bool -- fill in isodd with a definition
    if n == 0 then return false
    else return iseven(n - 1) end
end
```

For simple cases where all mutually recursive functions can be defined together, we allow both functions to be defined simultaneously:

```
terra iseven(n : int) : bool
    if n == 0 then return true
    else return isodd(n - 1) end
end

and terra isodd(n : int) : bool -- ‘and’ causes both definitions to occur together
    if n == 0 then return false
    else return iseven(n - 1) end
end
```

Performing typechecking lazily also provides several advantages. Forward declarations of functions, such as the declaration of `isodd`, do not have to have type annotations making them easier to maintain compared to languages such as C++ where symbols must be declared (*forward* declarations), and their type provided as well (traditional declaration). Furthermore, user-defined `struct` types do not need all their methods specified before being used in a Terra function, and are only required when the function is first typechecked. Compared to eager typechecking, it might seem like lazy checking introduces more times when errors are reported since errors can be reported when a function is first run. However, even if we performed type checking eagerly, we still might get *linking* errors when a function is first run if a function it referred to was never defined.

Though typechecking is performed lazily, we still allow code generation to happen during this stage through type-macros. Since type-macros are functions defined separately from Terra code and called like functions, they do not have access to the lexical environment where they are used so they can be delayed until typechecking without their behavior changing when a variable in that environment is mutated.

CHAPTER 4
FORMALIZING CODE GENERATION WITH TERRA CORE

To make the interaction between Lua and Terra precise, we formalize the essence of both languages focusing on how Terra functions are created, compiled, and called during the evaluation of a Lua program and in the presence of side-effects. This formalism will illustrate how some of the design decisions discussed in the previous chapter are implemented.

4.1 TERRA CORE

The calculus, called Terra Core, is equipped with a big step operational semantics. Evaluation starts in Lua ($\xrightarrow{L}$). When a Terra term is encountered it is specialized ($\xrightarrow{S}$), evaluating any escapes in the term to produce concrete Terra terms. Specialized Terra functions can then be executed ($\xrightarrow{T}$). We distinguish between Lua expressions e, Terra expressions $\dot{\mathsf{e}}$, and specialized Terra expressions $\underline{\dot{\mathsf{e}}}$ (we use a dot to distinguish Terra terms from Lua terms, and a bar to indicate a Terra term is specialized). For simplicity we model Lua as an imperative language with first-class functions and Terra as a purely functional language. A namespace Γ maps variables (x) to addresses a, and a store S maps addresses to Lua values v. The namespace Γ serves as the *value* environment of Lua (resolving variables to values, v), and the *syntactic* environment of Terra specialization (resolving variables to specialized Terra terms $\underline{\dot{\mathsf{e}}}$, which are a subset of Lua values). In contrast, Terra is executed in a separate environment ($\dot{\Gamma}$).

The Lua (Core) syntax is given in the following table:

$$
\begin{array}{rcl}
\mathsf{e} & ::= & \mathsf{b} \mid \dot{\mathsf{T}} \mid \mathsf{x} \mid \mathtt{let}\,\mathsf{x} = \mathsf{e}\,\mathtt{in}\,\mathsf{e} \mid \mathsf{x} := \mathsf{e} \mid \mid \mathsf{e}(\mathsf{e}) \mid \\
 & & \mathtt{fun}(\mathsf{x})\{\mathsf{e}\} \mid \mathtt{tdecl} \mid \mathtt{ter}\ \mathsf{e}(\mathsf{x} : \mathsf{e}) : \mathsf{e}\,\{\,\dot{\mathsf{e}}\,\} \mid \grave{}\dot{\mathsf{e}} \\
\mathsf{v} & ::= & \mathsf{b} \mid l \mid \dot{\mathsf{T}} \mid \langle \Gamma, \mathsf{x}, \mathsf{e} \rangle \mid \underline{\dot{\mathsf{e}}} \\
\dot{\mathsf{T}} & ::= & \dot{\mathsf{B}} \mid \dot{\mathsf{T}} \rightarrow \dot{\mathsf{T}}
\end{array}
$$

A Lua expression can be a base value (b), a Terra type expression ($\dot{\mathsf{T}}$), a variable (x), a scoped variable definition ($\mathtt{let}\ \mathsf{x} = \mathsf{e}\ \mathtt{in}\ \mathsf{e}$), an assignment ($\mathsf{x} := \mathsf{e}$), a function call $\mathsf{e}(\mathsf{e})$, a Lua function ($\mathtt{fun}(\mathsf{x})\{\mathsf{e}\}$), or a quoted Terra expression ($\grave{}\dot{\mathsf{e}}$). We separate declaration and definition of Terra functions to allow for recursive functions. A Terra function declaration (tdecl) creates a new address for a Terra function, while a Terra definition ($\mathtt{ter}\ \mathsf{e}_1(\mathsf{x} : \mathsf{e}_2) : \mathsf{e}_3\ \{\ \dot{\mathsf{e}}\ \}$) fills in the declaration at address e_1. For example, the following declares and defines a Terra function, storing it in x:

$$\mathtt{let}\ \mathsf{x} = \mathtt{ter}\ \ \mathtt{tdecl}(\mathsf{x}_2 : \mathtt{int}) : \mathtt{int}\ \{\ \mathsf{x}_2\ \}\ \mathtt{in}\ \mathsf{x}$$

Alternatively, tdecl creates just a declaration that can be defined later:

$$\mathtt{let}\ \mathsf{x} = \mathtt{tdecl}\ \mathtt{in}\ \mathtt{ter}\ \ \mathsf{x}(\mathsf{x}_2 : \mathtt{int}) : \mathtt{int}\ \{\ \mathsf{x}_2\ \}$$

In real Terra code, a Terra definition will create a declaration if it does not already exist. Lua values range over base types (b), addresses of Terra functions (l), Terra types ($\dot{\mathsf{T}}$), Lua closures ($\langle \Gamma, \mathsf{x}, \mathsf{e} \rangle$) and specialized Terra expressions ($\underline{\dot{\mathsf{e}}}$). The syntax of Terra terms is defined as follows:

$$\dot{\mathsf{e}} \ ::= \ \mathsf{b} \mid \mathsf{x} \mid \dot{\mathsf{e}}(\dot{\mathsf{e}}) \mid \mathtt{tlet}\,\mathsf{x} : \mathsf{e} = \dot{\mathsf{e}}\,\mathtt{in}\,\dot{\mathsf{e}} \mid [\mathsf{e}]$$

A Terra expression is either a base type, a variable, a function application, a let statement, or a Lua escape (written $[\mathtt{e}]$). The syntax of specialized terms is given next:

$$\boxed{\underline{\dot{\mathtt{e}}} \;::=\; \mathtt{b} \mid \underline{\dot{\mathtt{x}}} \mid \underline{\dot{\mathtt{e}}}(\underline{\dot{\mathtt{e}}}) \mid \mathtt{tlet}\; \underline{\dot{\mathtt{x}}} : \dot{\mathtt{T}} = \underline{\dot{\mathtt{e}}}\; \mathtt{in}\; \underline{\dot{\mathtt{e}}} \mid l}$$

In contrast to an unspecialized term, a specialized Terra term does not contain escape expressions, but can contain Terra function addresses (l). The let statement must assign Terra types to the bound variable and variables are replaced with specialized Terra variables $\underline{\dot{\mathtt{x}}}$.

The judgment $\mathtt{e}\ \Sigma_1 \xrightarrow{L} \mathtt{v}\ \Sigma_2$ describes the evaluation of a Lua expression. It operates over an environment Σ consisting of Γ, S, and a Terra function store F which maps addresses (l) to Terra functions. Terra functions can be defined ($\langle \underline{\dot{\mathtt{x}}}, \dot{\mathtt{T}}, \dot{\mathtt{T}}, \underline{\dot{\mathtt{e}}} \rangle$), or undefined ($\bullet$). Figure 4.1 defines the Lua evaluation rules. We use two notational shortcuts:

$$\begin{aligned}
\Sigma_1[\mathtt{x} \leftarrow \mathtt{v}] = \Gamma_2, S_2, F &\qquad \text{when } \Sigma_1 = \Gamma_1, S_1, F \wedge \Gamma_2 = \Gamma_1[\mathtt{x} \leftarrow a] \wedge \\
&\qquad S_2 = S_1[a \leftarrow \mathtt{v}] \wedge a \text{ fresh} \\
\Sigma \leftarrow \Gamma_1 = \Gamma_1, S, F &\qquad \text{when } \Sigma = \Gamma_2, S, F
\end{aligned}$$

Rule LTDecl creates a new Terra function at address l and initializes it as undefined ($\bullet$). Rule LTDefn takes an undefined Terra function ($\mathtt{e}_1$) and initializes it. First, $\mathtt{e}_2$ and $\mathtt{e}_3$ are evaluated as Lua expressions to produce the type of the function, $\dot{\mathtt{T}}_1 \to \dot{\mathtt{T}}_2$. The body, $\dot{\mathtt{e}}$, is specialized. During specialization, Terra variables ($\mathtt{x}$) are renamed to new symbols ($\underline{\dot{\mathtt{x}}}$) to ensure hygiene. Renaming has been previously applied in staged-programming [73] and hygienic macro expansion [4]. In the case of LTDefn, we generate a fresh name $\underline{\dot{\mathtt{x}}}$ for the formal parameter $\mathtt{x}$, and place it in the environment. Variable $\mathtt{x}$ will be bound to the value $\underline{\dot{\mathtt{x}}}$ in the scope of any Lua code evaluated during specialization of the function. During specialization, Rule SVar will replace uses of $\mathtt{x}$ in Terra code with the value of $\mathtt{x}$ in the environment.

Rule LTApp describes how to call a Terra function from Lua. The actual parameter $\mathtt{e}_2$ is evaluated. The Terra function is then typechecked. Semantically, typechecking occurs every time a function is run. In practice, we cache the result of typechecking. For simplicity, Terra

$$\mathsf{v}\ \Sigma\ \xrightarrow{L}\ \mathsf{v}\ \Sigma \quad (\text{LVal}) \qquad \frac{\Sigma = \Gamma, S, F}{\mathsf{x}\ \Sigma\ \xrightarrow{L}\ S(\Gamma(\mathsf{x}))\ \Sigma}\ (\text{LVar})$$

$$\frac{\mathsf{e}_1\ \Sigma_1\ \xrightarrow{L}\ \mathsf{v}_1\ \Sigma_2 \qquad \Sigma_2 = \Gamma, S, F \qquad \mathsf{e}_2\ \Sigma_2[\mathsf{x} \leftarrow \mathsf{v}_1]\ \xrightarrow{L}\ \mathsf{v}_2\ \Sigma_3}{\mathtt{let}\ \mathsf{x} = \mathsf{e}_1\ \mathtt{in}\ \mathsf{e}_2\ \Sigma\ \xrightarrow{L}\ \mathsf{v}_2\ (\Sigma_3 \leftarrow \Gamma)} \quad (\text{LLet})$$

$$\frac{\mathsf{e}\ \Sigma\ \xrightarrow{L}\ \mathsf{v}\ \Gamma, S, F \qquad \Gamma(\mathsf{x}) = a}{\mathsf{x} := \mathsf{e}\ \Sigma\ \xrightarrow{L}\ \mathsf{v}\ \Gamma, S[a \leftarrow \mathsf{v}], F} \quad (\text{LAsn})$$

$$\frac{\Sigma = \Gamma, S, F}{\mathtt{fun}(\mathsf{x})\{\mathsf{e}\}\ \Sigma\ \xrightarrow{L}\ \langle \Gamma, \mathsf{x}, \mathsf{e} \rangle\ \Sigma} \quad (\text{LFun})$$

$$\frac{\begin{array}{c}\mathsf{e}_1\ \Sigma_1\ \xrightarrow{L}\ \langle \Gamma_1, \mathsf{x}, \mathsf{e}_3 \rangle\ \Sigma_2 \qquad \mathsf{e}_2\ \Sigma_2\ \xrightarrow{L}\ \mathsf{v}_1\ \Gamma_2, S, F \\ a\ \mathtt{fresh} \qquad \mathsf{e}_3\ \Gamma_1[\mathsf{x} \leftarrow a], S[a \leftarrow \mathsf{v}_1], F\ \xrightarrow{L}\ \mathsf{v}_2\ \Sigma_3\end{array}}{\mathsf{e}_1(\mathsf{e}_2)\ \Sigma_1\ \xrightarrow{L}\ \mathsf{v}_2\ (\Sigma_3 \leftarrow \Gamma_2)} \quad (\text{LApp})$$

$$\frac{l\ \mathtt{fresh} \qquad \Sigma = \Gamma, S, F}{\mathtt{tdecl}\ \Sigma\ \xrightarrow{L}\ l\ \Gamma, S, F[l \leftarrow \bullet]} \quad (\text{LTDecl})$$

$$\frac{\begin{array}{c}\mathsf{e}_1\ \Sigma_1\ \xrightarrow{L}\ l\ \Sigma_2 \qquad \mathsf{e}_2\ \Sigma_2\ \xrightarrow{L}\ \dot{\mathsf{T}}_1\ \Sigma_3 \qquad \mathsf{e}_3\ \Sigma_3\ \xrightarrow{L}\ \dot{\mathsf{T}}_2\ \Sigma_4 \\ \Sigma_4 = \Gamma_1, S_1, F_1 \qquad \underline{\dot{\mathsf{x}}}\ \mathtt{fresh} \\ \dot{\mathsf{e}}\ \Sigma_4[\mathsf{x} \leftarrow \underline{\dot{\mathsf{x}}}]\ \xrightarrow{S}\ \underline{\dot{\mathsf{e}}}\ \Gamma_2, S_2, F_2 \qquad F_2(l) = \bullet\end{array}}{\mathtt{ter}\ \mathsf{e}_1(\mathsf{x} : \mathsf{e}_2) : \mathsf{e}_3\ \{\ \dot{\mathsf{e}}\ \}\ \Sigma_1\ \xrightarrow{L}\ l\ \Gamma_1, S_2, F_2[l \leftarrow \langle \underline{\dot{\mathsf{x}}}, \dot{\mathsf{T}}_1, \dot{\mathsf{T}}_2, \underline{\dot{\mathsf{e}}} \rangle]} \quad (\text{LTDefn})$$

$$\frac{\dot{\mathsf{e}}\ \Sigma_1\ \xrightarrow{S}\ \underline{\dot{\mathsf{e}}}\ \Sigma_2}{`\dot{\mathsf{e}}\ \Sigma_1\ \xrightarrow{L}\ \underline{\dot{\mathsf{e}}}\ \Sigma_2} \quad (\text{LTQuote})$$

$$\frac{\begin{array}{c}\mathsf{e}_1\ \Sigma_1\ \xrightarrow{L}\ l\ \Sigma_2 \qquad \mathsf{e}_2\ \Sigma_2\ \xrightarrow{L}\ \mathsf{b}_1\ \Sigma_3 \\ \Sigma_3 = \Gamma, S, F \qquad F(l) = \langle \underline{\dot{\mathsf{x}}}, \dot{\mathsf{T}}_1, \dot{\mathsf{T}}_2, \underline{\dot{\mathsf{e}}} \rangle \qquad \mathsf{b}_1 \in \dot{\mathsf{T}}_1 \\ [\underline{\dot{\mathsf{x}}} : \dot{\mathsf{T}}_1], [l : \dot{\mathsf{T}}_1 \to \dot{\mathsf{T}}_2], F_2 \vdash \underline{\dot{\mathsf{e}}} : \dot{\mathsf{T}}_2 \qquad \underline{\dot{\mathsf{e}}}\ [\underline{\dot{\mathsf{x}}} \leftarrow \mathsf{b}], F\ \xrightarrow{T}\ \mathsf{b}_2\end{array}}{\mathsf{e}_1(\mathsf{e}_2)\ \Sigma_1\ \xrightarrow{L}\ \mathsf{b}_2\ \Sigma_3} \quad (\text{LTApp})$$

Figure 4.1: The rules $\xrightarrow{L}$ for evaluating Lua expressions.

$$\mathsf{b}\ \Sigma \xrightarrow{S} \mathsf{b}\ \Sigma \qquad \text{(SBAS)}$$

$$\frac{\overset{\bullet}{\mathsf{e}}_1\ \Sigma_1 \xrightarrow{S} \underline{\overset{\bullet}{\mathsf{e}}}_1\ \Sigma_2 \qquad \overset{\bullet}{\mathsf{e}}_2\ \Sigma_2 \xrightarrow{S} \underline{\overset{\bullet}{\mathsf{e}}}_2\ \Sigma_3}{\overset{\bullet}{\mathsf{e}}_1(\overset{\bullet}{\mathsf{e}}_2)\ \Sigma_1 \xrightarrow{S} \underline{\overset{\bullet}{\mathsf{e}}}_1(\underline{\overset{\bullet}{\mathsf{e}}}_2)\ \Sigma_3} \qquad \text{(SAPP)}$$

$$\frac{\begin{array}{c}\mathsf{e}\ \Sigma_1 \xrightarrow{L} \overset{\bullet}{\mathsf{T}}\ \Sigma_2 \qquad \overset{\bullet}{\mathsf{e}}_1\ \Sigma_2 \xrightarrow{S} \underline{\overset{\bullet}{\mathsf{e}}}_1\ \Sigma_3 \qquad \underline{\overset{\bullet}{\mathsf{x}}}\ \text{fresh} \\ \Sigma_3 = \Gamma, S, F \qquad \overset{\bullet}{\mathsf{e}}_2\ \Sigma_3[\mathsf{x} \leftarrow \underline{\overset{\bullet}{\mathsf{x}}}] \xrightarrow{S} \underline{\overset{\bullet}{\mathsf{e}}}_2\ \Sigma_4\end{array}}{\mathtt{tlet}\ \mathsf{x} : \mathsf{e} = \overset{\bullet}{\mathsf{e}}_1\ \mathtt{in}\ \overset{\bullet}{\mathsf{e}}_2\ \Sigma_1 \xrightarrow{S} \mathtt{tlet}\ \underline{\overset{\bullet}{\mathsf{x}}} : \overset{\bullet}{\mathsf{T}} = \underline{\overset{\bullet}{\mathsf{e}}}_1\ \mathtt{in}\ \underline{\overset{\bullet}{\mathsf{e}}}_2\ (\Sigma_4 \leftarrow \Gamma)} \qquad \text{(SLET)}$$

$$\underset{\text{(SESC)}}{\frac{\mathsf{e}\ \Sigma_1 \xrightarrow{L} \underline{\overset{\bullet}{\mathsf{e}}}\ \Sigma_2}{[\mathsf{e}]\ \Sigma_1 \xrightarrow{S} \underline{\overset{\bullet}{\mathsf{e}}}\ \Sigma_2}} \qquad \underset{\text{(SVAR)}}{\frac{[\mathsf{x}]\ \Sigma_1 \xrightarrow{S} \underline{\overset{\bullet}{\mathsf{e}}}\ \Sigma_2}{\mathsf{x}\ \Sigma_1 \xrightarrow{S} \underline{\overset{\bullet}{\mathsf{e}}}\ \Sigma_2}}$$

Figure 4.2: The rules $\xrightarrow{S}$ for specializing Terra expressions.

$$\mathsf{b}\ \overset{\bullet}{\Gamma}, F \xrightarrow{T} \mathsf{b} \qquad \text{(TBAS)}$$

$$l\ \overset{\bullet}{\Gamma}, F \xrightarrow{T} l \qquad \text{(TFUN)}$$

$$\underline{\overset{\bullet}{\mathsf{x}}}\ \overset{\bullet}{\Gamma}, F \xrightarrow{T} \overset{\bullet}{\Gamma}(\underline{\overset{\bullet}{\mathsf{x}}}) \qquad \text{(TVAR)}$$

$$\frac{\underline{\overset{\bullet}{\mathsf{e}}}_1\ \overset{\bullet}{\Gamma}, F \xrightarrow{T} \mathsf{v}_1 \qquad \underline{\overset{\bullet}{\mathsf{e}}}_2\ \overset{\bullet}{\Gamma}[\underline{\overset{\bullet}{\mathsf{x}}} \leftarrow \mathsf{v}_1], F \xrightarrow{T} \mathsf{v}_2}{\mathtt{tlet}\ \underline{\overset{\bullet}{\mathsf{x}}} : \overset{\bullet}{\mathsf{T}} = \underline{\overset{\bullet}{\mathsf{e}}}_1\ \mathtt{in}\ \underline{\overset{\bullet}{\mathsf{e}}}_2\ \overset{\bullet}{\Gamma}, F \xrightarrow{T} \mathsf{v}_2} \qquad \text{(TLET)}$$

$$\frac{\begin{array}{c}\underline{\overset{\bullet}{\mathsf{e}}}_1\ \overset{\bullet}{\Gamma}, F \xrightarrow{T} l \qquad \underline{\overset{\bullet}{\mathsf{e}}}_2\ \overset{\bullet}{\Gamma}, F \xrightarrow{T} \mathsf{v}_1 \\ F(l) = \langle \underline{\overset{\bullet}{\mathsf{x}}}, \overset{\bullet}{\mathsf{T}}_1, \overset{\bullet}{\mathsf{T}}_2, \underline{\overset{\bullet}{\mathsf{e}}}_3 \rangle \qquad \underline{\overset{\bullet}{\mathsf{e}}}_3\ \overset{\bullet}{\Gamma}[\underline{\overset{\bullet}{\mathsf{x}}} \leftarrow \mathsf{v}_1], F \xrightarrow{T} \mathsf{v}_2\end{array}}{\underline{\overset{\bullet}{\mathsf{e}}}_1(\underline{\overset{\bullet}{\mathsf{e}}}_2)\ \overset{\bullet}{\Gamma}, F \xrightarrow{T} \mathsf{v}_2} \qquad \text{(TAPP)}$$

Figure 4.3: The rules $\xrightarrow{T}$ for evaluating Terra expressions.

$$\frac{\hat{F}(l) = \overset{\bullet}{\mathsf{T}}}{\hat{\Gamma}, \hat{F}, F \vdash l : \overset{\bullet}{\mathsf{T}}} \qquad \text{(TYFUN1)}$$

$$\frac{l \notin \hat{F} \qquad F(l) = \langle \mathsf{x}, \overset{\bullet}{\mathsf{T}}_1, \overset{\bullet}{\mathsf{T}}_2, \underline{\overset{\bullet}{\mathsf{e}}} \rangle \qquad [\mathsf{x} : \overset{\bullet}{\mathsf{T}}_1], \hat{F}[l : \overset{\bullet}{\mathsf{T}}_1 \to \overset{\bullet}{\mathsf{T}}_2], F \vdash \underline{\overset{\bullet}{\mathsf{e}}} : \overset{\bullet}{\mathsf{T}}_2}{\hat{\Gamma}, \hat{F}, F \vdash l : \overset{\bullet}{\mathsf{T}}_1 \to \overset{\bullet}{\mathsf{T}}_2} \qquad \text{(TYFUN2)}$$

Figure 4.4: Typing rules for references to Terra functions.

Core only allows values b of base types to be passed and returned from Terra functions (full Terra is less restricted).

Figure 4.2 defines judgment $\dot{\mathsf{e}}\ \Sigma_1 \xrightarrow{S} \underline{\dot{\mathsf{e}}}\ \Sigma_2$ for specializing Terra code, which evaluates all embedded Lua expressions in type annotations and escape expressions. Similar to LTDEFN, rule SLET generates a unique name $\underline{\dot{\mathsf{x}}}$ to ensure hygiene. Rule SESC evaluates escaped Lua code; it splices the result into the Terra expression if the resulting value is in the subset of values that are Terra terms $\underline{\dot{\mathsf{e}}}$ (e.g., a variable $\underline{\dot{\mathsf{x}}}$ or base value b). Variables in Terra can refer to variables defined in Lua and in Terra; they behave as if they are escaped, as defined by Rule SVAR. If x is a variable defined in Terra code and renamed $\underline{\dot{\mathsf{x}}}$ during specialization, then rule SVAR will just produce $\underline{\dot{\mathsf{x}}}$ (assuming no interleaving mutation of x).

Figure 4.3 presents the judgment $\underline{\dot{\mathsf{e}}}\ \dot{\Gamma}, F \xrightarrow{T} \mathsf{v}$ for evaluating specialized Terra expressions. These expressions can be evaluated independently from the Lua store S, and do not modify F, but are otherwise straightforward. A Terra function is typechecked right before it is run (LTAPP) with the judgment $\hat{\Gamma}, \hat{F}, F \vdash \underline{\dot{\mathsf{e}}} : \dot{\mathsf{T}}$, where $\hat{\Gamma}$ is the typing environment for variables and $\hat{F}$ is the typing environment for Terra function references (F is the Terra function store from before). The rules (omitted for brevity) are standard, except for the handling of Terra function references l. If a Terra function l_1 refers to another Terra function l_2, then l_2 must be typechecked when typechecking l_1. The rules for handling these references in the presence of mutually recursive functions are shown in Figure 4.4. They ensure all functions that are in the connected component of a function are typechecked before the function is run.

4.2 DESIGN DECISIONS AS REFLECTED IN TERRA CORE

Terra Core formalizes some of the features that were presented informally in the previous chapter. Here we describe how these features work in the calculus.

4.2.1 *Terra entities are first-class*

Terra types and expressions are Lua values. This is illustrated in Terra Core. Terra *syntax* $\underline{\dot{\mathsf{e}}}$ is one type of Lua *value*, along with Terra types, $\dot{\mathsf{T}}$. As an example, a Lua Core function can manipulate Terra types. The Lua function x_3 will generate a Terra identity function for any given type:

$$\begin{aligned}&\texttt{let}\ \mathsf{x}_3 = \texttt{fun}(\mathsf{x}_1)\{\texttt{ter tdecl}(\mathsf{x}_2 : \mathsf{x}_1) : \mathsf{x}_1\ \{\ \mathsf{x}_2\ \}\}\ \texttt{in}\\ &\mathsf{x}_3(\texttt{int})(1)\end{aligned}$$

Here we call it with `int`, which will result in the specialized Terra function $\langle \underline{\dot{\mathsf{x}}}, \texttt{int}, \texttt{int}, \underline{\dot{\mathsf{x}}}\rangle$. Chapters 6 and 7 will expand on this formalism, showing how we can use Lua to generate arbitrary Terra types.

4.2.2 *Hygiene and shared lexical environment*

Terra Core illustrates how we provide a shared lexical environment and hygiene. The evaluation of Lua code and the specialization of Terra code share the same lexical environment Γ and store S. This environment and store always map variables x to Lua values v. This example illustrates the shared environment:

$$\begin{aligned}&\texttt{let}\ \mathsf{x}_1 = 0\ \texttt{in}\\ &\texttt{let}\ \mathsf{x}_2 =\grave{}\ (\texttt{tlet}\ \mathsf{y}_1 : \texttt{int} = 1\ \texttt{in}\ \mathsf{x}_1)\ \texttt{in}\\ &\texttt{let}\ \mathsf{x}_3 = \texttt{ter tdecl}(\mathsf{y}_2 : \texttt{int}) : \texttt{int}\ \{\ \mathsf{x}_2\ \}\ \texttt{in}\ \mathsf{x}_3\end{aligned}$$

The specialization of the quoted `tlet` expression occurs in the surrounding Lua environment, so Rule SVAR will evaluate x_1 to 0. This results in the specialized expression:

$$\texttt{tlet}\ \underline{\dot{\mathsf{y}}}_1 : \texttt{int} = 1\ \texttt{in}\ 0$$

This Terra expression will be stored as a Lua value in x_2. Since the Terra function refers to x_2, specialization will result in the following Terra function:

$$\langle \underline{\dot{\mathsf{y}}}_2, \texttt{int}, \texttt{int}, \texttt{tlet}\ \underline{\dot{\mathsf{y}}}_1 : \texttt{int} = 1\ \texttt{in}\ 0\rangle$$

Furthermore, during specialization variables introduced by Terra functions and Terra `let` expressions are bound in the shared lexical environment. Consider this example:

$$\begin{array}{l}\texttt{let}\ \mathsf{x}_1 = \mathsf{fun}(\mathsf{x}_2)\{\grave{}\,\texttt{tlet}\ \mathsf{y} : \mathsf{int} = 0\ \texttt{in}\ [\mathsf{x}_2]\}\ \texttt{in} \\ \texttt{let}\ \mathsf{x}_3 = \texttt{ter tdecl}(\mathsf{y} : \mathsf{int}) : \mathsf{int}\ \{\ [\mathsf{x}_1(\mathsf{y})]\ \}\ \texttt{in}\ \mathsf{x}_3\end{array}$$

The variable y on line 2 is introduced by the Terra function definition. It is referenced by the Lua expression inside the escape ($[\mathsf{x}_1(\mathsf{y})]$). The variable y is then passed as an argument to Lua function x_1, where it is spliced into a `tlet` expression.

When Terra variables are introduced into the environment, they are given fresh names to ensure hygiene. For example, without renaming, x_3 would specialize to the following, causing the `tlet` expression to unintentionally capture y:

$$\langle \mathsf{y}, \mathsf{int}, \mathsf{int}, \texttt{tlet}\ \mathsf{y} : \mathsf{int} = 1\ \texttt{in}\ \mathsf{y}\rangle$$

To avoid this, rules LTDEFN and SLET generate fresh names for variables declared in Terra expressions. In this case, the LTDEFN will generate a fresh name $\underline{\dot{\mathsf{y}}}_1$ for the argument y binding it into the shared environment ($\Sigma[\mathsf{y} \leftarrow \underline{\dot{\mathsf{y}}}_1]$), and SLET will similarly generate the fresh name $\underline{\dot{\mathsf{y}}}_2$ for the `tlet` expression. Since y on line 2 has the *value* $\underline{\dot{\mathsf{y}}}_1$ during specialization, the variable x_2 will get the value $\underline{\dot{\mathsf{y}}}_1$, and x_3 will specialize to the following, avoiding the unintentional capture:

$$\langle \underline{\dot{\mathsf{y}}}_1, \mathsf{int}, \mathsf{int}, \texttt{tlet}\ \underline{\dot{\mathsf{y}}}_2 : \mathsf{int} = 1\ \texttt{in}\ \underline{\dot{\mathsf{y}}}_1\rangle$$

4.2.3 *Eager specialization with lazy typechecking*

Eager specialization prevents mutations in Lua code from changing the meaning of a Terra function between when it is defined and when it is used. Terra Core helps illustrate where problems can arise. For instance, consider the following example (we use the syntax $\mathsf{e}; \mathsf{e}$ as sugar for $\texttt{let}\ \_ = \texttt{e}\ \texttt{in}\ \texttt{e}$):

$$
\begin{array}{l}
\texttt{let}\ \mathsf{x}_1 = 0\ \texttt{in} \\
\texttt{let}\ \mathsf{y} = \texttt{ter tdecl}(\mathsf{x}_2 : \texttt{int}) : \texttt{int}\ \{\ \mathsf{x}_1\ \}\ \texttt{in} \\
\mathsf{x}_1 := 1; \\
\mathsf{y}(0)
\end{array}
$$

Since specialization is performed eagerly, the statement $\mathsf{y}(0)$ will evaluate to 0. In contrast, if specialization were performed *once* lazily, then it would capture the value of x_1 the first time y is called and keep that value for the rest of the program, which would lead to surprising results (e.g., if y were used before $\mathsf{x}_1 := 1$ then it would always return 0, otherwise it would always return 1). Alternatively, we could re-specialize (and hence re-compile) the function when a Lua value changes, but this behavior could lead to large compiler overheads that would be difficult to track down.

Eager specialization requires all symbols used in a function to be declared before the function's definition, which can be problematic for mutually recursive functions. In order to support recursive functions with eager specialization, we separate the declaration and definition of Terra functions:

$$
\begin{array}{l}
\texttt{let}\ \mathsf{x}_2 = \texttt{tdecl}\ \texttt{in} \\
\texttt{let}\ \mathsf{x}_1 = \texttt{ter tdecl}(\mathsf{y} : \texttt{int}) : \texttt{int}\ \{\ \mathsf{x}_2(\mathsf{y})\ \}\ \texttt{in} \\
\texttt{ter}\ \mathsf{x}_2(\mathsf{y} : \texttt{int}) : \texttt{int}\ \{\ \mathsf{x}_1(\mathsf{y})\ \}; \\
\mathsf{x}_1(0)
\end{array}
$$

In contrast to specialization, typechecking is performed lazily. In Terra Core, it would be possible to perform typechecking eagerly if declarations also had types. For instance, in our previous example we could typecheck x_1 when it is defined if x_2 was given a type during declaration. However, even though x_1 would typecheck, we would still receive a *linking* error if $\mathsf{x}_1(0)$ occurred before the definition of x_2. Terra Core helps illustrate that performing typechecking eagerly would not reduce the number of places an error might occur for function x_1. Furthermore, unlike specialization where the result can change arbitrarily depending on the Lua state, the result of typechecking and linking x can only change monotonically from a type-error to success as the functions it references are defined (it can also stay as a type-error if the function is actually ill-typed). This property follows from the fact that Terra functions can be defined, but not re-defined by Rule LTDEFN.

4.2.4 *Separate evaluation of Terra code*

After Terra code is compiled, it can run independently from Lua. This behavior is captured in Terra Core by the fact that Terra expressions are evaluated independently from the environment Γ and the store S, as illustrated by this example:

$$\begin{array}{l}
\mathtt{let}\ \mathsf{x}_1 = 1\ \mathtt{in} \\
\mathtt{let}\ \mathsf{y} = \mathtt{ter}\ \mathtt{tdecl}(\mathsf{x}_2\ : \mathtt{int}) : \mathtt{int}\ \{\ \mathsf{x}_1\ \}\ \mathtt{in} \\
\mathsf{x}_1 := 2; \mathsf{y}(0)
\end{array}$$

The Terra function will specialize to $\langle \overset{\bullet}{\underline{\mathsf{x}}}, \mathtt{int}, \mathtt{int}, 1\rangle$, so the function call will evaluate to the value 1, despite x_1 being re-assigned to 2. An alternative design would allow Terra evaluation to directly refer to x_1, which is the behavior of MetaOCaml [72]. Terra evaluation would then depend on Γ and S, coupling the runtimes of Lua and Terra together. As previously discussed, this coupling would make it difficult to reason about performance, or run Terra on GPUs and embedded devices.

CHAPTER 5
BUILDING DSLS

To evaluate Terra, we use it to reimplement a number of multi-language applications and compare our implementations with existing approaches. We present evidence that the design decisions of Terra make the implementations simpler to engineer compared to existing work while achieving high performance.

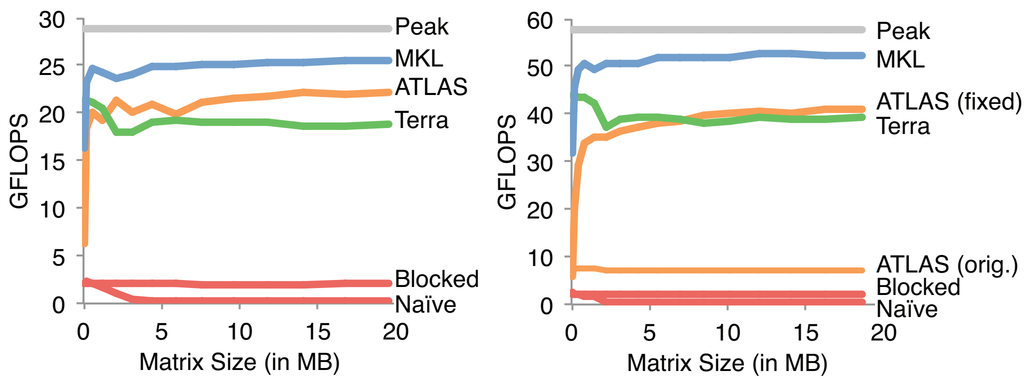
5.1 TUNING DGEMM

BLAS routines like double-precision matrix multiply (DGEMM) are used in a wide range of applications and form a basis for many of the algorithms used in high-performance scientific computing. However, their performance is dependent on characteristics of the machine such as cache sizes, vector length, or number of floating-point machine registers. In our tests, a naïve DGEMM can run over 65 times slower than the best-tuned algorithm.

The ATLAS project [77] was created to maintain high performance BLAS routines via auto-tuning. To demonstrate Terra's usefulness in auto-tuning high-performance code, we implemented a version of matrix multiply, the building block of level-3 BLAS routines. We restrict ourselves to the case $C = AB$, with both A and B stored non-transposed, and base our optimizations on those of ATLAS [77]. ATLAS breaks down a matrix multiply into smaller operations where the matrices fit into L1 cache. An optimized kernel for L1-sized multiplies is used for each operation. Tuning DGEMM involves choosing good block sizes, and generating optimized code for the L1-sized kernel. We found that a simple two-level blocking scheme worked well. To generate the L1-sized kernel, we use staging to implement several optimizations. We implement register-blocking of the inner-most loops, where a

```
function genkernel(NB, RM, RN, V,alpha)
  local vector_type = vector(double,V)
  local vector_pointer = &vector_type
  local A,B,C = symbol("A"),symbol("B"),symbol("C")
  local mm,nn = symbol("mn"),symbol("nn")
  local lda,ldb,ldc = symbol("lda"),symbol("ldb"),symbol("ldc")
  local a,b = symmat("a",RM), symmat("b",RN)
  local c,caddr = symmat("c",RM,RN), symmat("caddr",RM,RN)
  local k = symbol("k")
  local loadc,storec = terralib.newlist(),terralib.newlist()
  for m = 0, RM-1 do for n = 0, RN-1 do
      loadc:insert(quote
        var [caddr[m][n]] = C + m*ldc + n*V
        var [c[m][n]] =
          alpha * @vector_pointer([caddr[m][n]])
      end)
      storec:insert(quote
        @vector_pointer([caddr[m][n]]) = [c[m][n]]
      end)
  end end
  local calcc = terralib.newlist()
  for n = 0, RN-1 do
    calcc:insert(quote
      var [b[n]] = @vector_pointer(&B[n*V])
    end)
  end
  for m = 0, RM-1 do
    calcc:insert(quote
      var [a[m]] = vector_type(A[m*lda])
    end)
  end
  for m = 0, RM-1 do for n = 0, RN-1 do
      calcc:insert(quote
        [c[m][n]] = [c[m][n]] + [a[m]] * [b[n]]
      end)
  end end
  return terra([A] : &double, [B] : &double, [C] : &double,
               [lda] : int64,[ldb] : int64,[ldc] : int64)
    for [mm] = 0, NB, RM do
      for [nn] = 0, NB, RN*V do
        [loadc];
        for [k] = 0, NB do
          prefetch(B + 4*ldb,0,3,1);
          [calcc];
          B,A = B + ldb,A + 1
        end
        [storec];
        A,B,C = A - NB,B - ldb*NB + RN*V,C + RN*V
      end
      A,B,C = A + lda*RM, B - NB, C + RM * ldb - NB
end end end
```

Figure 5.1: Parameterized Terra code that generates a matrix-multiply kernel optimized to fit in L1.



(a) DGEMM Performance (b) SGEMM Performance

Figure 5.2: Performance of matrix multiply using different libraries as a function of matrix size. Size reported is the total footprint for both input and output matrices. All matrices are square.

block of the output matrix is stored in machine registers; we vectorize this inner-most loop using vector types; and we use prefetch intrinsics to optimize non-contiguous reads from memory.

The code that implements our L1-sized kernel is shown in Figure 5.1. It is parameterized by the blocksize (`NB`), the amount of the register blocking in 2 dimensions (`RM` and `RN`), the vector size (`V`), and a constant (`alpha`) which parameterizes the multiply operation, `C = alpha*C + A*B`. When generating code with a parameterizable number of variables (e.g., for register blocking) it is sometimes useful to selectively violate hygiene. Terra provides the function `symbol`, equivalent to LISP's `gensym`, which generates a globally unique identifier that can be used to define and refer to a variable that will not be renamed. We use it on lines 4–9 to generate the intermediate variables for our computation (`symmat` generates a matrix of symbols). On lines 10–20, we generate the code to load the values of `C` into registers (`loadc`), and the code to store them back to memory (`storec`). Lines 21–31 load the `A` and `B` matrices, and lines 32–36 generate the unrolled code to perform the outer product(`calcc`). We compose these pieces into the L1-sized matrix multiply function (lines 37–51). The full matrix-multiply routine (not shown) calls the L1-sized kernel for each block of the multiply.

In Lua, we wrote an auto-tuner that searches over reasonable values for the parameters (`NB`, `V`, `RA`, `RB`), JIT-compiles the code, runs it on a user-provided test case, and choses the best-performing configuration. Our implementation is around 200 lines of code.

```
void diffuse(int N, int b, float* x, float* x0, float* tmp,
             float diff, float dt ){
  int i, j, k; float a=dt*diff*N*N;
  for (k = 0; k<= iter; k++){
    for (j = 1; j <= N; j++)
      for (i = 1; i <= N; i++)
        tmp[IX(i,j)] = (x0[IX(i,j)] + a*(x[IX(i-1,j)]+
          x[IX(i+1,j)]+x[IX(i,j-1)]+x[IX(i,j+1)]))/(1+4*a);
    SWAP(x,tmp);
  }
}
```

```
function diffuse ( x, x0, diff, dt )
  local a=dt*diff*N*N
  for k=0,iter do
    x = (x0+a*(x(-1,0)+x(1,0)+x(0,-1)+x(0,1)))/(1+4*a)
  end
  return x,x0
end
```

Figure 5.3: A kernel from a real-time fluid solver written in C (top) compared to Darkroom (bottom).

We evaluate the performance by comparing to ATLAS and Intel's MKL on a single core of an Intel Core i7-3720QM. ATLAS 3.10 was compiled with GCC 4.8. Figure 5.2 shows the results for both double- and single- precision. For DGEMM, the naïve algorithm performs poorly. While blocking the algorithm does improve its performance for large matrices, it runs at less than 7% of theoretical peak GFLOPs for this processor. In contrast, Terra performs within 20% of the ATLAS routine, over 60% of peak GFLOPs of the core, and over 65 times faster than the naïve unblocked code. The difference between Terra and ATLAS is likely caused by a register spill in Terra's generated code that is avoided in ATLAS's generated assembly. Terra is also competitive with Intel's MKL, which is considered state-of-the-art. For SGEMM, Terra outperforms the unmodified ATLAS code by a factor of 5 because ATLAS incurs a transition penalty from mixing SSE and AVX instructions. Once this performance bug is fixed, ATLAS performs similarly to Terra.

ATLAS is built using Makefiles, C, and assembly programs generated with a custom preprocessor. The Makefiles orchestrate the creation and compilation of the code with different parameters. Code generation is accomplished through a combination of preprocessors and cross-compilation written in C. Auto-tuning is performed using a C harness for timing. Different stages communicate through the file system.

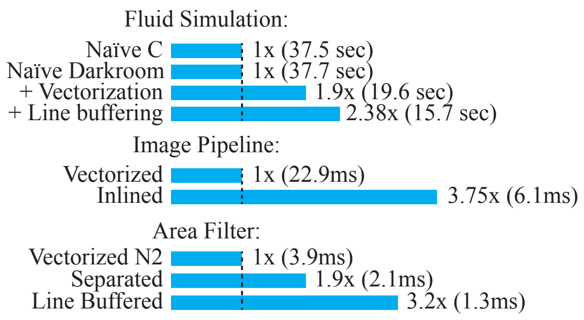


Figure 5.4: Speedup from choosing different Darkroom schedules. All results on Intel Core i7-3720QM, 1024x1024 floating point pixels.

The design of Terra allows all of these tasks to be accomplished in one system and as a single process. Terra provides low-level features like vectors and prefetch instructions needed for high-performance. In contrast, ATLAS needed to target x86 directly, which resulted in a performance bug in SGEMM. Staging annotations made it easy to write parameterized optimizations like register unrolling without requiring a separate preprocessor. Interoperability through the FFI made it possible to generate and evaluate the kernels in the same framework. Finally, since Terra code can run without Lua, the resulting multiply routine can be written out as a library and used in other programs; or, for portable performance, it can be shipped with the Lua runtime and auto-tuning can be performed dynamically, something that is not possible with ATLAS.

5.2 Darkroom: A stencil DSL for images

To test Terra's suitability for DSL development, we created Darkroom[1], a DSL for 2D stencil computations on images. Stencil computations are grid-based kernels in which each value in the grid is dependent on a small local neighborhood. They are used in image processing and simulation. They present a number of opportunities for optimization, but when implemented like the C code in Figure 5.3, it is difficult to exploit the performance opportunities. For example, fusing two iterations of the outer loop in `diffuse` may reduce memory traffic, but testing this hypothesis can require significant code changes. Figure 5.3 shows the same `diffuse` operation written in Darkroom. Rather than specify loop nests directly, Darkroom programs are written using image-wide operators. For instance, `f(-1,0) + f(0,1)` adds the

[1]In our paper [23], this DSL was referred to as Orion.

image `f` translated by -1 in x to `f` translated by 1 in y. The offsets must be constants, which guarantees the function is a stencil.

We base our design on Halide [61], a language for the related domain of image processing. The user guides optimization by specifying a *schedule*. A Darkroom expression can be *materialized*, *inlined*, or *line buffered*. Materialized expressions are computed once and stored to main memory. Inlined expressions are recomputed once for each output pixel. Line buffering is a compromise in which computations are interleaved and the necessary intermediates are stored in a scratchpad. Additionally, Darkroom can vectorize any schedule using Terra's vector instructions. Being able to easily change the schedule is a powerful abstraction. To demonstrate this, we implemented a pipeline of four simple memory-bound point-wise image processing kernels (blacklevel offset, brightness, clamp, and invert). In a traditional image processing library, these functions would likely be written separately so they could be composed in an arbitrary order. In Darkroom, the schedule can be changed independently of the algorithm. For example, we can choose to inline the four functions, reducing the accesses to main memory by a factor of 4 and resulting in a 3.8x speedup.

To implement Darkroom, we use operator overloading on Lua tables to build Darkroom expressions. These operators build an intermediate representation (IR) suitable for optimization. The user calls `darkroom.compile` to compile the IR into a Terra function. We then use Terra's staging annotations to generate the code for the inner loop.

To test that the code generated by Terra performs well, we implemented an area filter and a fluid simulation. We compare each to equivalents hand-written in C. The area filter is a common image processing operation that averages the pixels in a 5x5 window. Area filtering is separable, so it is normally implemented as a 1-D area filter first in Y then in X. We compare against a hand-written C implementation with results in Figure 5.4. Given a schedule that matches the C code, Darkroom performs similarly, running 10% faster. Enabling vectorization in Darkroom yields a 2.8x speedup over C, and then line buffering between the passes in Y and X yields a 3.4x speedup. Explicit vectors are not part of standard C, and writing line-buffering code is tedious and breaks composability, so these optimizations are not normally done when writing code by hand.

We also implemented a simple real-time 2D fluid simulation based on an existing C implementation [66]. We made small modifications to the reference code to make it suitable to a stencil language. We converted the solver from Gauss-Seidel to Gauss-Jacobi so that images are not modified in place and use a zero boundary condition since our implementation does not yet support more complicated boundaries. We also corrected a performance bug in the code caused by looping over images in row-major order that were stored in column-major order. We compare against the corrected version. With a matching schedule, Darkroom performs the same as reference C. Enabling 4-wide vectorization results in a 1.9x speedup over the matching code, making each materialized operation memory bound. Finally, line buffering pairs of the iterations of the diffuse and project kernels yielded a 1.25x speedup on the vectorized code, or a 2.3x total speedup over the reference C code.

A number of features of Terra facilitated the implementation of Darkroom. High-level features of Lua made it easy to express transformations on the Darkroom IR. Terra's built-in support of vector types made it easy to vectorize the compiler by simply changing scalar types into vectors. Backwards compatibility with C allowed us to link to an existing library for loading images. The FFI made it possible to use Lua to implement non-performance-critical code such as the kernel scheduler, saving development time. Furthermore, the fluid simulation that we ported included a semi-Lagrangian advection step, which is not a stencil computation. In this case, we were able to allow the user to pass a Terra function to do the necessary computation, and easily integrate this code with generated Terra code. This interoperability would have been more difficult to accomplish with a stand-alone compiler.

In contrast to Darkroom, Halide requires three different languages to provide the same functionality as Darkroom. It uses C++ for the frontend, ML for manipulating the IR, and LLVM for code generation [61]. When compared running the same deblurring algorithm both Darkroom and Halide perform almost equivalently (.35 and .37 seconds respectively) [37]. From our experience implementing Darkroom, using Lua to stage Terra code accomplishes the same tasks, but results in a simpler architecture.

5.3 LISZT: A DSL FOR PHYSICAL SIMULATION

Liszt is a physical simulation language whose optimizations were covered in Section 2.1.3. The original version of Liszt [22] produced high performance code, but it is architected out of several technologies, making it hard to maintain. It uses Scala as a frontend and for its offline compiler implementation. The compiler uses a source-to-source translation step to convert Liszt's IR into C++ code. A runtime is written separately in C++, which provides common functionality that the generated code could use, such as allocating fields or loading meshes.

We implemented a new version of Liszt. We use Lua to write the compiler transformations and manage the DSL's state, and we use Terra as the target of code generation and as the language for writing high-performance libraries for working with mesh data structures.

Using Lua and Terra as a replacement for two separate technologies removed complexity from Liszt's design. The original offline compiler required careful design decisions that were easier to make in the Lua-Terra version. Frequently, we needed to decide whether code for part of the Liszt program should go in a runtime function or be part of the code generated by the compiler. Writing to Liszt's fields are a good illustration of the problem. Fields are parameterized over the type of mesh element (e.g., vertex or edge), and the kind of data it stores (e.g., float, int, vector). Writes might be a simple assignment, or a reduction operator such as `+=`. Liszt runs on both CPUs or GPUs and the field representation on each is different so the code to write the fields differs. One option is to generate different code for all of these options in the code generator. This design provides maximum control and efficiency, but results in lots of implementation details being embedded as strings in the source-to-source code generator. Another option is to make all of these decisions in a runtime function that is provided dynamic tags describing what operation to perform. This design is easier to read and understand, but can incur runtime overhead if the offline compiler does not inline and optimize the calls into the runtime (a form of partial evaluation).

After multiple iterations on design, the best solution for generating field writes is a mix of the two extremes. But the original architecture of Liszt made these decisions difficult to prototype. Changing where code was created required moving it from string templates in a Scala file to C functions in the runtime. The alternative in Lua-Terra is much simpler. Different fragments of code are first-class Lua objects, and can be managed by the code generator. A runtime function that made decisions about fields would exist as an object in the Lua environment. If the programmer wanted to change it to a template that specialized some decisions, that object could simply be replaced with a Terra macro that generated the appropriate code. Furthermore, since both exist in the same process, it is easy to debug the result by introspection. It is also impossible to generate code that refers to a non-existent runtime function, something that could easily happen when generating string templates.

Since the original Liszt implementation generates C++, and Terra uses the same compiler pipeline as traditional C compilers, the performance of Terra-generated code is comparable to the original version of Liszt. Using Lua-Terra instead of separate processes also makes it easier to integrate Liszt into existing applications, and to use externally created libraries of C code. Our interface for language extensions also makes it easier to embed the Liszt frontend in Lua, allowing the offloading of many language features to the Lua runtime.

5.4 QUICKSAND: A DSL FOR PROBABILISTIC PROGRAMMING

Probabilistic programming languages (PPLs) are a general-purpose modeling tool for artificial intelligence, machine learning, and statistics [33, 46]. Probabilistic programs define probability distributions: the program makes random choices (such as flipping a weighted coin), and running the program produces a sample from the marginal distribution implied by those choices. By conditioning the output of the program on a predicate, programmers can pose interesting queries of their models.

A *universal* PPL such as the probabilistic Scheme dialect Church can describe any Turing-complete, stochastic process, including recursive processes and distributions with infinite support [33]. These languages are expressive, but their implementations are slower than equivalent hand-coded models. One reason for the performance gap is algorithmic: inference algorithms (i.e., implementations of conditioning semantics) must be general-purpose and cannot easily optimize for the statistical traits of specific models. Another reason is computational: probabilistic inference is a numerically-intensive task, but existing universal PPL implementations are high-level and dynamically-typed, making it harder to control their performance.

Addressing this second problem, we implemented a Church-style universal PPL, Quicksand, as a library in Terra. The query “What is the chance that a patient has lung cancer,

```
id = 0
function pfn(fn)
    return macro(function(self, ...)
        id = id + 1
        local args = terralib.newlist({...})
        local argIntermediates = args:map(
            function(a) return symbol(a:gettype()) end)
        return quote
            var [argIntermediates] = [args]
            callsiteStack:push(id)
            var result = fn([argIntermediates])
            callsiteStack:pop()
        in result end
    end)
end
```

Figure 5.5: A higher-order Lua function that wraps a Terra function for use in MCMC inference.

given that she has a cough?" in a simplistic medical diagnosis model is expressed in our Terra-based language as:

```
terra()
    var lungcancer = flip(0.01)
    var cold = flip(0.2)
    var cough = lungcancer or cold
    condition(cough)
    return lungcancer
end
```

In contrast to other DSLs, Quicksand is written in Terra functions directly using special type-macros like `flip` and `condition`. The functionality of these operators is difficult to implement directly in a low-level language such as C++ because it lacks sophisticated meta-programming. This is illustrated in a few places in Quicksand's design.

First, the Markov Chain Monte Carlo (MCMC) inference algorithm used to sample from conditioned programs requires that every random choice in the program be given an *address* that is uniquely determined by the choice's structural position in program execution traces [78]. Typically, addresses are managed with a global stack of function callsites: every function call pushes a unique ID on entry and pops the stack on exit. When a random choice is invoked, the sequence of IDs on the stack plus the number of times that sequence has occurred in the current program execution uniquely identifies the random choice.

We implement this behavior using type-macros in Figure 5.5. The Lua function `pfn` wraps a Terra function with macro code that manages the address stack. Critically, the macro is executed during typechecking, so each *callsite* of the function receives a distinct ID.

Managing random choice addresses is more complicated in non-staged languages. Previous implementations of universal PPLs use either a custom interpreter or a source-to-source transformation [33, 78]. Interpretation is too slow for our performance goals, and source transformation requires a complete parse of the program, which is a steep price to pay for embedding a universal PPL in most low-level languages.

We must also define the behavior of primitive random choices (e.g., `flip`). These constructs must sample a value from an underlying probability distribution (e.g., Bernoulli), compute the probability of their values under the distribution, and propose changes to their values for use in MCMC inference. Our implementation exposes a Lua function `makeRandomChoice` for this purpose:

```
flip = makeRandomChoice(
    -- Sampling function
    terra(p: double) return (rand() < p) end,
    -- Probability function
    terra(val: bool, p: double)
        if val then return log(p) else return log(1-p) end
    end,
    -- Proposal function
    terra(currval: bool, p: double) return (not currval) end)
```

Internally, `makeRandomChoice` defines a `RCRecord` type that records the value and parameters for each invocation of this random choice. The entries of `RCRecord` are determined programmatically by examining the argument and return types of the sampling function. `makeRandomChoice` returns a `Pfn` that constructs an instance of `RCRecord` and stores it in a global table mapping random choice addresses to values.

Because Terra allows reflection on functions to drive the generation of new types, we can abstract the details of the random choice as a single library routine. The user only needs to provide three functions, which can be overloaded to handle random choices taking different parameter types. Without the ability to generate code with arbitrary types the behavior of every random choice would be implemented independently for each set of parameter types, or would require sophisticated parameterized typing.

We evaluate the performance of our Quicksand implementation on three example programs: conditioned sampling of a length-100 sequence from a Hidden Markov Model with 9 hidden states and 10 possible observations (HMM); learning the parameters of a three-component Gaussian Mixture Model from 1000 data points (GMM); and simulating

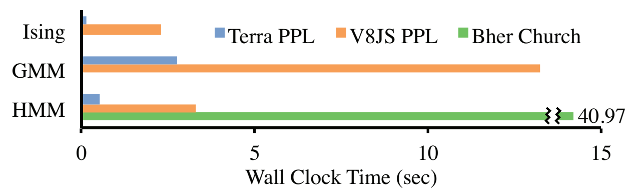


Figure 5.6: Performance of different probabilistic programming languages on three example tasks. Javascript runs an order of magnitude faster than Bher, and our Terra implementation runs an order of magnitude faster than that.

a 1000-site one-dimensional Ising model (Ising). All examples were run for 5000 iterations of MCMC. We compare performance to that of Bher (a compiled implementation of Church [78]) and a probabilistic dialect of V8 Javascript actively used in teaching. We only evaluate Church on the first example because the other two require language primitives not available in Church.

Results are shown in Figure 5.6. The Javascript implementation is an order of magnitude faster than Bher on the HMM example. This difference is due in part to V8's optimization engine, and also to the overhead of Bher's purely functional data structures (particularly the trie it uses for random choice addressing). Our Terra implementation is, on average, an additional order of magnitude faster than Javascript. Terra's static typing allows up-front compilation of much more monomorphic code, reducing the number of virtual function calls as well as boxing/unboxing overhead.

CHAPTER 6
GENERATING TYPES WITH EXOTYPES

We have already seen that for code generators, the two-language design produces code that is fast and concise. Many of these applications, such as Darkroom or Liszt, include their own frontend syntax and focus on optimizing the composition of language operators. But others, such as ATLAS or Quicksand, expose a simpler API to the end users. One way to support this kind of simpler library is to build code generation into the way Terra's type system works.

Object-oriented systems associate method calls with particular types, providing a way to abstractly work with data. For instance, in the method invocation `student:serializeTo(buffer)`, the programmer can still use the serialization method even if they are not aware of the details of the implementation. This abstraction layer of types and methods allows the library writer to use as much complexity as they want when implementing the behavior. For instance, they can use reflection and code generation to compile the serialization method on demand. This complexity does not change the interface that the user of the library sees.

If a library writer can use types that do code generation internally, it allows users of the types who are not necessarily aware of the techniques of staged programming to still benefit from the approach. This technique is sometimes referred to as an *active* library since the types in the library are internally doing dynamic compilation [20].

Unfortunately, active libraries are not widely used. One reason is that it is difficult to deploy them in current languages. Static and dynamic approaches to type checking of these libraries present different challenges. Statically-typed languages make it difficult or impossible to do arbitrary computation on types. Typically, the amount of computation on types is limited to what can be expressed in the type system itself, for example inheritance or parameterization. Even in cases like template meta-programming in C++ where many computations are possible, the interface for doing sophisticated computation is so complex that it is not typically used. Active libraries for many statically-typed languages instead resort to pre-processors like Google's protocol buffers [34], which provide serialization for C++ by generating a library of C++ types based on an external description.

In contrast, dynamically-typed languages often provide the flexibility necessary to do reflection and provide generic implementations of types. One reason is that many dynamic languages such as Lisp, Python, or Lua support so-called *meta-object protocols*, meaning there is a mechanism for the user to programmatically modify the semantics and implementation of user-defined types [3, 40, 45]. Higher-level policies such as inheritance or accessor permissions can be defined on top of these mechanisms, giving the programmer great flexibility in defining object behavior, including providing their own generated behavior.

However, when an object's behavior and in-memory representation are defined dynamically, it is difficult to perform some optimizations, resulting in performance losses. For instance, in Section 6.3, we implement a microbenchmark of an `Array` object that forwards methods to each of its elements. The JIT-compiled Lua version runs 18 times slower than equivalent C++ code due to object boxing and dynamic dispatch. JIT compilers can optimize some dynamic patterns [21, 38], but it is difficult to know if a pattern will result in high-performance code.

The two-language design of Lua and Terra is nether a fully dynamically-typed nor statically-typed language. We can use this hybrid design to combine the advantages of each system to make it possible to do arbitrary computation on types but avoid the overhead of runtime checks. This design makes it easy to create active libraries of concise, highly optimized, and composable types.

Our approach is to extend the typical typechecking process that statically-typed languages undergo to include user-defined computation on types. When Terra's typechecker needs to know how a user-defined type should behave (e.g. what does it mean to call method `serializeTo` on object `student`?), it calls a *Lua* function provided by the definer of the type. That function can do arbitrary computation on the type, generating code that implements the behavior, or describing other aspects of the type such as its memory layout. The result is passed back to the typechecker. We call these types *exotypes* since they are defined in a Lua

API that is external to Terra itself, which differs from other statically-typed languages that include their own statements for defining types.

Exotypes combine meta-object protocols with multi-stage programming to give the programmer more control over the code's performance. Rather than define an object's behavior as a function that is evaluated at runtime, an exotype describes the behavior using staged programming during typechecking. This design allows the programmer to optimize behavior and memory layout before any instances of the object are used.

As a concrete example, consider joining two different employee databases that both contain an "employee ID" field. To implement the join efficiently in a low-level language, the structure of both databases must be described in code beforehand. In a dynamic language, the structure can be deduced at runtime by reading a database schema, but the programmer has less control over the layout of the objects. They may be boxed, adding an extra level of indirection, and their fields may be stored in hash-tables rather than linearly in memory. With exotypes, the database structure can be read at runtime while retaining a compact object layout. The first stage of the program reads the database schema and generates exotypes with fixed, compact data layouts. With the object layout known, the second stage actually compiles and runs the join, exploiting the compact layout of the generated types to store objects unboxed and access them with simple pointer arithmetic.

We introduce the concept of an exotype and present a concrete implementation based on programmatically-defined *properties* queried during typechecking. We show that high-level type features such as type constructors can be created with exotypes. In the next chapter, we present formal semantics for the way that Terra's typechecker interacts with Lua when evaluating exotype properties. When specified in a well-behaved manner, independently-defined type constructors can be composed. In Chapter 8, we evaluate the use of exotypes in several performance-critical scenarios: class-systems, database layout, automatic differentiation, serialization, and dynamic assembly.

6.1 Meta-object protocols

Modern dynamic languages allow programmatic definition of object behavior. For instance, Python provides *metaclasses* which can override the default behaviors of method definition and invocation, and CLOS allows for the dynamic specification of all behavior of objects using so-called meta-object protocols [45, 3]. The Lua language uses a meta-object protocol based on *metatables* to extend the normal semantics of objects [40]. Metatables are Lua tables containing functions that define new semantics for default behaviors. For instance, we can change the behavior of the table indexing operator `obj.field` by setting the `__index` field in a metatable:

```
local myobj = {}
setmetatable(myobj,
  { __index = function(self,field) return field end })
print(myobj.somefield) -- prints "somefield"
```

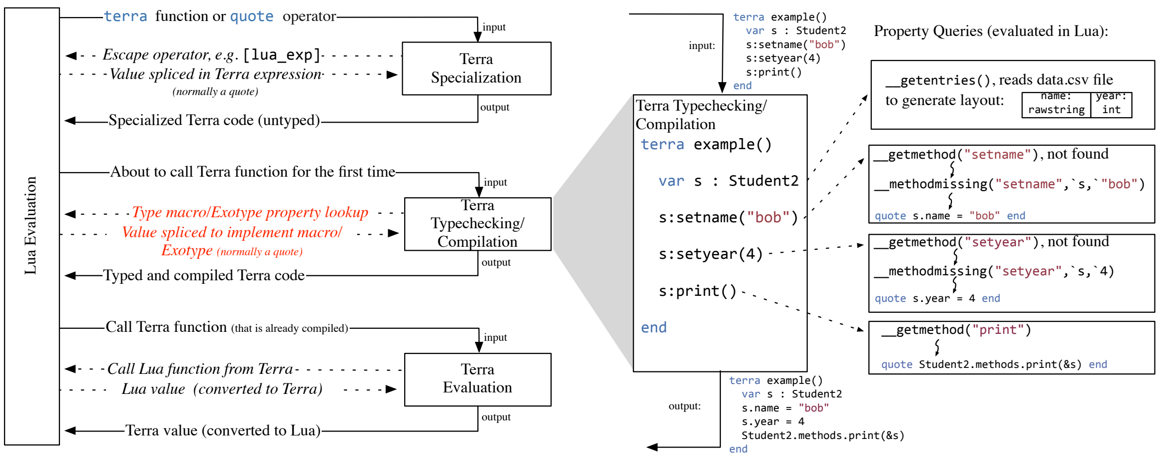
When the expression `myobj.somefield` is evaluated, the Lua interpreter will look for the key `"somefield"` in the `myobj` table. If the key does not exist, it will instead call the `__index` function of `myobj`'s metatable passing the object and the missing key as arguments and returning the result as the value of the original expression. Metatables also contain other functions that similarly define other behaviors such as function application and arithmetic operators.

We use a meta-object protocol defined using Lua tables to describe the behavior of exotypes. However, the behaviors in exotypes are expressed using staged programming and queried before the code that uses the objects is compiled. While most meta-object protocols are applied dynamically, some, such as those in Open-C++, are applied statically during compilation [14]. In these systems, no new types are defined at runtime. Exotypes blend the two approaches. New types can be created and compiled as the program runs, but since exotype behavior is described with staged programming of a low-level language (Terra), the programmer retains control over low-level representation and implementation.

6.2 Exotypes interface

We define an *exotype* as a tuple of functions which will be called during typechecking to define the layout and behavior of a user-defined type:

$$(() \to \texttt{MemoryLayout}) * (\texttt{Op}_0 \to \texttt{Quote}) * \ldots * (\texttt{Op}_n \to \texttt{Quote})$$


Lua Evaluation
terra function or quote operator
input
Escape operator, e.g. [lua_exp]
Value spliced in Terra expression (normally a quote)
Terra Specialization
output
Specialized Terra code (untyped)
About to call Terra function for the first time
input
Type macro/Exotype property lookup
Value spliced to implement macro/ Exotype (normally a quote)
Terra Typechecking/ Compilation
output
Typed and compiled Terra code
Call Terra function (that is already compiled)
input
Call Lua function from Terra
Lua value (converted to Terra)
Terra Evaluation
output
Terra value (converted to Lua)
input:
Terra Typechecking/ Compilation
output:
Property Queries (evaluated in Lua):
__getentries(), reads data.csv file to generate layout:
name: rawstring
year: int
__getmethod("setname"), not found
__getmethod("setyear"), not found
__getmethod("print")

```
terra example()
  var s : Student2
  s:setname("bob")
  s:setyear(4)
  s:print()
end
```

```
__methodmissing("setname",`s,`"bob")
quote s.name = "bob" end

__methodmissing("setyear",`s,`4)
quote s.year = 4 end

quote Student2.methods.print(&s) end
```

```
terra example()
  var s : Student2
  s.name = "bob"
  s.year = 4
  Student2.methods.print(&s)
end
```

Figure 6.1: Phases of evaluation of a Lua-Terra program with exotypes (left), and an example of the typechecking process for the `Student2` exotype (right). New interactions used to implement exotypes are highlighted in red.

PROPERTY	OPERATION
`__getentries()` Defines the in-memory representation of `T` as a list of named fields.	`obj.myfield`
`__getmethod(name)` Gets the static implementation of `name` for type `T`.	`obj:mymethod()`
`__add(lhs,rhs)` Defines the behavior of the + operator on the object (`lhs` or `rhs` has type `T`).	`obj + 1`
`__cast(from,to,exp)` Defines how to convert expression `exp` of type `from` to type `to`.	`[int](obj)`
`__apply(arg1,...,argN)` Defines how to apply an instance of `T` to a list of arguments.	`obj(1.0, true)`

Figure 6.2: A selection of exotype properties and example operations that cause the type checker to invoke them.

The first function computes the in-memory *layout* of a type, which is specified with `MemoryLayout`. The remaining functions describe the *semantics* of the type when it appears in a primitive operation of the language such as a method invocation, binary operator, or cast. Given an instance of a primitive operation ($\texttt{Op}_i$), the corresponding function returns a `Quote`, a concrete expression that implements the instance. These functions are evaluated by the typechecker whenever it encounters an operation on an exotype and may reference and modify program state. We call these functions *property functions* and the results of these functions *properties* of the type.

In the remainder of this section, we discuss an implementation of exotypes that uses Lua to define types in Terra, and type-macros to define the type's behavior. In this implementation, exotypes are the only mechanism for creating user-defined Terra types. Terra's `struct` syntax to define types is implemented via desugaring to exotypes. An exotype is created via an API call in Lua:

```
Student = terralib.types.newstruct()
```

Property functions are defined in the type's `metamethods` table. The in-memory layout of a type is specified with `__getentries`. For instance, the `Student` type can be defined to have two fields – a name and a class year:

```
Student.metamethods.__getentries = function()
    return { {field = "name", type = rawstring },
             {field = "year", type = int} }
end
```

Since properties are defined programmatically, we are not limited to explicitly enumerated entries. A type can define its layout by querying the layout of another type or accessing external information. For example, we define `Student2` by reading a comma-separated value file of student data and inferring the type of the fields from the data:

```
Student2.metamethods.__getentries = function()
    local file = io.open("data.csv","r")  --e.g. name,year
    local titles = split(",",file:read("*line"))
    local data   = split(",",file:read("*line"))
    local entries = {}
    for i,field in ipairs(titles) do --loop over entries in titles
        --is the data a string or an integer?
        local type = tonumber(data[i]) and int or rawstring
        entries[i] = { field = field, type = type}
    end
    return entries
end
```

To keep code concise, we include simple default implementations. For `__getentries` in type `T`, we return `T.entries`, which is automatically populated by Terra's `struct` statement:

```
struct Student3 { name : rawstring, year : int }
```

Semantics are also specified with property functions in the `metamethods` table. When a method is invoked on an instance of type `T` (e.g., `t:mymethod(arg)`), the implementation of the method is defined using the `__getmethod` property of type `T`. By default it looks up the method in the table `T.methods`. But if the method being invoked does not exist, it will query the `__methodmissing` property. Here is an example of using methods with the `Student2` type, in which we use the `__methodmissing` property to create setter methods (e.g., `setname`) for each field.

```
Student2.methods.print = terra(self : &Student2)
    C.printf("%s in year %d\n",self.name,self.year)
end
Student2.metamethods.__methodmissing = macro(function(name,self,arg)
    local field = string.match(name,"set(.*)")
    if field then
      return quote self.[field] = arg end
    end
    error("unknown method: "..name)
end)
```

The typechecking process for example code that uses `Student2` is illustrated in Figure 6.1 (right). When the typechecker sees an instantiation of the type, it queries `__getentries` to get its memory layout. For `Student2` this will load the `data.csv` file to determine the layout. If the typechecker sees a call to a method like `setname` which is not in `Student2.methods`, then the `__getmethod` property will query the `__methodmissing` property for its behavior. Since `__methodmissing` is defined as a type-macro, it will be evaluated during the typechecking process to produce a Terra quotation that implements the behavior.

Other properties define the behaviors for built-in operators (e.g., `__add` for `+`), and how to convert between user-defined types (`__cast`). Figure 6.2 lists some common properties and the operations they define.

Defining behavior programmatically allows generic behaviors to be expressed concisely. For instance, we can interface with externally-defined class systems. Objective-C is an extension to C that adds objects similar in behavior to Smalltalk. We can embed Objective-C objects by creating an exotype `ObjC` wrapper as shown in Figure 6.3. When the typechecker sees a method called on an `ObjC` object (e.g., `obj:init(1)`), its `__methodmissing` property will insert code to call the Objective-C runtime API, e.g: `C.objc_msgSend(obj.handle,init_name,1)`

Staging of properties gives the programmer control over the performance of the type. Since behavior can be defined in type-macros that are evaluated during typechecking, information known statically can be pre-computed. In the `ObjC` type, the method name `sel` is

```
  C = terralib.includec("objc/message.h") --include ObjC runtime
  struct ObjC { handle : &C.objc_object } --define wrapper for ObjC types
  ObjC.metamethods.__methodmissing = macro(function(sel,obj,...)
      local arguments = {...}
5     local sel = C.sel_registerName(sanitizeSelector(sel,#arguments))
      --generate expression to implement method call 'sel'
      return `ObjC { C.objc_msgSend([obj].handle,[sel],[arguments]) }
  end)
```

Figure 6.3: Embedding Objective-C objects in Terra using exotypes.

known during typechecking, so we can pre-compute the Objective-C method selector (line 5), which makes the expression executed at runtime (line 7) faster.

Furthermore, `__getentries` provides low-level control of the memory layout of a type similar to that of C structs. Types are laid out linearly by default and can also be overlapped in unions. This control allows `__getentries` to describe more efficient memory layouts. We implemented `Student` objects individually, as if they were entries in a *row*-oriented database. But we can change `__getentries` to store students as entries in a *column*-oriented database which may be more efficient.

6.3 Example: Array(T)

Exotypes allow us to express the generic behavior of objects in a way similar to meta-object protocols in dynamic languages while still achieving the performance of low-level languages. Consider how to represent the type constructor `Array(·)`, which takes a type `T` and produces a new type, `Array(T)`, that holds a collection of `T`s. In addition, for each method `m` of `T`, `Array(T)` has its own method `m` that invokes the original `m` on each member of the array. This is a *Proxy* design pattern [32], where methods on one object are forwarded to methods on another. This specification is different from that in most functional languages, where this behavior is implemented as a higher-order `map` function but is widely used in array-based languages such as APL and R to concisely operate on collections.

Proxy patterns are difficult to express generically in many modern languages. Despite the fact that proxies are a common pattern in object-oriented computing, programmers using C++ or Java must duplicate the method names of the object in the proxy. Some languages have added more advanced features to support proxies and other design patterns.

```
  local function createmethod(self,methodname)
      local impl = function(self,...)
          for i,element in ipairs(self.data) do
              element[methodname](element,...)
5         end
      end
      self[methodname] = impl --cache result
      return impl
  end
10 local function createarray()
      local arr = {data = terralib.newlist()}
      return setmetatable(arr, { __index = createmethod })
  end
```

Figure 6.4: An implementation of the array constructor in Lua

For instance, Hannemann and Kiczales show that AspectJ can automate some aspects of proxies, but not in a way that is generically reusable [36]. Expressing proxies in Scala requires the use of advanced features such as its macro facilities and `Dynamic` type trait that were only added in version `2.10` of the language [9].

In contrast, proxies are relatively easy to implement in dynamic languages. As an example, `Array(·)` can be implemented concisely using metatables in Lua as shown in Figure 6.4. When a missing method of an array is referenced, the metatable's `__index` field (line 12) causes `createmethod` to be called, which generates an implementation for the method (line 2) that loops over the objects in the array forwarding the method call to each.

While simple, it is hard to control the performance of the object. Methods are looked up dynamically on each object in the array. The objects themselves are boxed, limiting memory locality. We implemented a micro-benchmark that invoked a simple counter function on each member of an array object and evaluated it using LuaJIT, a state-of-the-art tracing JIT [2]. Despite tracing and compiling the loop, it still performs 18 times slower than hand-written C++ that implements the proxy by hand due to the overhead of guards and unboxing of objects.

With exotypes, we can specify `Array(·)` almost as concisely as the Lua code, but staging of its behavior and layout allows us to remove the inefficiencies, as shown in Figure 6.5. The `__methodmissing` type macro performs a similar purpose to the `createmethod` function in the Lua example. It generates the code that loops over elements of the array and forwards

```
   Array = memoize(function(T)
       local struct ArrayImpl {
           data : &T,
           N : int
 5     }
       ArrayImpl.metamethods.__methodmissing =
       macro(function(methodname,selfexp,...)
           local args = terralib.newlist {...}
           return quote
10           var self = selfexp
             for i = 0,self.N do
               self.data[i]:[methodname]([args])
             end
           end
15     end)
       --other implementation like :init()
       return ArrayImpl
   end)
```

Figure 6.5: An implementation of the array constructor using Exotypes in Terra

the method call, but it does so during typechecking rather than evaluation. Furthermore, the declaration of `ArrayImpl` provides a concrete layout for the type before it is compiled.

Since the layout of `ArrayImpl` is described before compilation, we can store the array's `T` objects unboxed rather than as an array of pointers. Furthermore, the forwarded calls to `methodname` on line 12 are resolved at compile time and inlined. In our micro-benchmark, this staging allowed us to generically generate the same code as the hand-written C++ proxy, and run at the same speed.

6.4 Composability

It is possible to apply type constructors such as `Array(·)` to other programmatically-defined exotypes. To support composability, we allow an implementation of an exotype to query the properties of other exotypes. However, type hierarchies are often recursive, making it possible for multiple types to mutually depend on properties of each other. Consider the following `Tree` type:

```
struct Tree { data : int, children : Array(Tree) }
Tree.methods.print = terra(self : &Tree) : {}
    print(self.data)
    self.children:print()
end
```

The type `Tree` contains a type `Array(Tree)` that is defined programmatically with the `Array(·)` function using `Tree` itself as an argument. The in-memory layout of `Tree` depends on the layout of `Array(Tree)`. Similarly, the method `print` of `Array(Tree)` depends on the method `print` in `Tree`. (The layout of `Array(Tree)` does not depend on `Tree`, since it only stores a pointer to it.)

Since the dependencies are for different properties, there is no actual circular dependence between the types. The required type information can be determined from the specification in steps. First, the in-memory layout is determined for `Array(Tree)` and then `Tree`, then the two methods are defined, first `print` for `Tree` and finally `print` for `Array(Tree)`.

The layout and method definitions of `Array(Tree)` and `Tree` must be interleaved to avoid causing a cyclic dependence. In recursive cases similar to this example, eagerly defining all the properties of one type before another can introduce a false dependency. However, interleaving the definition of `Array(Tree)` and `Tree` makes it impossible to define a single type constructor function `Array(·)` that completely defines the properties of a new type. These type-constructors are desirable because they can be provided as libraries and composed with other exotypes without requiring the caller to understand the specific implementation.

Our interface to exotypes resolves this problem by defining each property separately as a *lazily* evaluated function. The compiler queries an individual property of a type only when it is needed during typechecking. Evaluating properties separately and lazily allows the compiler to interleave property queries from different types while allowing them to be specified entirely for one type in a single function. Since typechecking and compilation of functions are also performed lazily, type properties are not requested earlier than needed.

In an earlier version of Terra, types were described using eagerly built tables that specified the layout and methods, in addition to ad-hoc user-defined callbacks invoked by the typechecker. Our experience with issues causing subtle cycles while building type constructors such as `Array(·)` led us to replace this design with exotypes based on lazily queried properties.

Lazily queried properties also make it possible to create types that have an unbounded number of behaviors. For instance, the Objective-C wrapper object presented previously can respond to an unlimited number of messages. Despite being unbounded, these methods can compose with other type constructors. For instance, it is possible to make an `Array(ObjC)` that stores OS windows and call `windows:makeKeyAndOrderFront(nil)` to focus them:

```
var windows : Array(ObjC)
var data = array(NSWindow:alloc(),NSWindow:alloc())
windows.data, windows.N = data, 2
windows:initWithContentRect_styleMask_backing_defer(
  NSRect{{0,0},{10,10}},1,2,false)
windows:makeKeyAndOrderFront(nil)
```

In this case, the method `makeKeyAndOrderFront` is requested by the typechecker via the `__getmethod` property of `Array(ObjC)`, which will in turn query the `__getmethod` property of `ObjC` to generate the method call. If `Array` required all methods to be available up-front, these two types would not compose.

6.5 Related Work

Previous work has proposed several optimizations to improve the performance of meta-object protocols in dynamic languages. Most of this work has focused on dynamically-dispatched protocols. For instance, Self, a language based on prototypes that was a precursor to more general protocols, used polymorphic inline caching to optimize method dispatch [38]. Kiczales et al. propose a meta-object protocol for CLOS that uses memoization to speed up method dispatch [45]. These approaches normally still incur some runtime overhead (e.g., to check the cache). In contrast, exotypes are executed during staged compilation, which removes all runtime overhead that is not desired by the user.

There is some work on meta-object protocols that are evaluated statically [48, 14]. The most popular of these approaches provides a meta-object protocol for C++, OpenC++, based on source-to-source translation that allows a programmer to customize the semantics of C++ classes and functions using meta-classes that produce program fragments [14]. Since these methods are applied ahead-of-time, they do not add any runtime overhead, but require that all information used to create the type be available during compilation. In contrast, Terra is a staged programming language, so new exotypes can be introduced during the execution of a Lua-Terra program, and used in newly generated functions.

The original work on multi-stage programming such as MetaML or MetaOCaml focused on generating new code rather than new types [73, 72]. But other systems extend this work to object-oriented languages and provide some degree of staged type computation. Metaphor is a multi-stage language with support for type reflection on a built-in class system [55], and Ur [15] allows first-class computation of records and names based on principles from dependently-typed languages. Both try to guarantee that any code produced using staging will be type correct, making it difficult to generate some types whose semantics depend on dynamically provided data.

Other work on staged programming has focused on optimizations that can be applied to object representations. Rompf et al. propose a system that implements internal compiler passes that use staging to optimize the behavior and the layout of objects in an intermediate representation suited to parallel programming embedded in the Scala language [63]. Type signatures are originally described using Scala's class system, but then their representation and implementation can be optimized through staging. Exotypes additionally allow the original type signatures and behavior to be described programmatically.

Several industrial languages also implement systems similar to exotypes [71, 9]. F# allows *type providers* which can specify types and methods based on external data [71]. Like exotypes, these providers are queried lazily when an operation on the type is requested. However, the goal of type providers is to safely type complex data representations, so providers are normally compiled ahead-of-time rather than during program execution. Furthermore, they normally describe types on top of the CLR's object system rather than have the programmer describe their own low-level implementation of types. In the Scala language, the combination of compile-time macros and syntax sugar for supporting dynamic objects allows some types to be described programmatically. Similar to F#, this approach is normally applied ahead-of-time and built on top of JVM objects rather than a low-level language. Other solutions are tailored to specific problems such as Google's protocol buffers, which provides a language for generating types with serialization behavior in several target languages [34].

CHAPTER 7
FORMALIZATION OF EXOTYPES

The interaction between Terra and Lua during Terra's typechecking phase adds additional complexity to a typical typechecking process since it allows the execution of user-defined code. This chapter provides additional semantics for the behavior of Lua and Terra during this typechecking process. Because we want to reason about termination, we will use a small step semantics for Lua-Terra that is adapted from the semantics presented earlier.

Complexity in typechecking exotypes arises from the evaluation of property functions. Exotype property functions can directly query properties of other exotypes. Properties are also queried indirectly by generated code. For instance, `Array(ObjC)` generates code that calls a method on `ObjC`, querying its `__getmethod` property. It is possible for an exotype to request a property resulting in true cyclic property lookups. In this case, our system emits an error.[1] Furthermore, because properties are arbitrary programs they are not guaranteed to terminate and may have other undesirable behaviors.

In order to discuss some constraints that make properties well-behaved we formally define the typechecking and property evaluation process for exotypes in Terra. We also examine the consequences of relaxing these constraints to make exotypes more flexible.

Terra is evaluated in multiple phases. To keep our formalization of exotypes simple, we focus on the typechecking and evaluation of Terra code with exotype property queries. In particular, we omit the machinery for quotes, escapes, and specialization used to generate Terra code, and assume that an environment `P` that holds a mapping between exotypes and

[1]Other behavior is possible. A recursively requested property could initially return the value $\top$ and iterate property lookup until a fixed point is reached.

$$\frac{\Gamma(\dot{\mathsf{x}}) = \mathsf{T}}{\mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{x}} : \mathsf{T}} \qquad (\textsc{TyVar})$$

$$\frac{\mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{e}} : \mathsf{T} \to \mathsf{T}' \qquad \mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{e}}' : \mathsf{T}}{\mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{e}}\ \dot{\mathsf{e}}' : \mathsf{T}'} \qquad (\textsc{TyApp})$$

$$\frac{\mathsf{P}\ \ \Gamma[\dot{\mathsf{x}} \leftarrow \mathsf{T}] \vdash \dot{\mathsf{e}} : \mathsf{T}'}{\mathsf{P}\ \ \Gamma \vdash \lambda\dot{\mathsf{x}} : \mathsf{T}.\dot{\mathsf{e}} : \mathsf{T} \to \mathsf{T}'} \ (\textsc{TyFun}) \qquad \mathsf{P}\ \ \Gamma \vdash \mathsf{c}_{\mathsf{Int}} : \mathsf{Int}\ (\textsc{TyInt})$$

$$\frac{\mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{e}} : \mathsf{T} \qquad \mathsf{P}\ \ \mathsf{runprop}\ \emptyset\ \mathsf{prop}\ \mathsf{t}\ \mathsf{layout}\ \ \longrightarrow_{\mathsf{L}}^{*}\ \ \mathsf{P}\ \ \mathsf{T}}{\mathsf{P}\ \ \Gamma \vdash \mathsf{c}_{\mathsf{t}}\ \dot{\mathsf{e}} : \mathsf{t}} \qquad (\textsc{TyCtor})$$

$$\frac{\mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{e}} : \mathsf{t} \qquad \mathsf{P}\ \ \mathsf{runprop}\ \emptyset\ \mathsf{prop}\ \mathsf{t}\ \mathsf{layout}\ \ \longrightarrow_{\mathsf{L}}^{*}\ \ \mathsf{P}\ \ \mathsf{T}}{\mathsf{P}\ \ \Gamma \vdash \mathsf{unwrap}\ \dot{\mathsf{e}} : \mathsf{T}} \qquad (\textsc{TyUnwrap})$$

$$\frac{\begin{array}{c}\mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{e}} : \mathsf{t} \qquad \mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{e}}' : \mathsf{T}' \\ \mathsf{P}\ \ \mathsf{runprop}\ \emptyset\ \mathsf{prop}\ \mathsf{t}\ \mathsf{apply}\ \mathsf{T}'\ \ \longrightarrow_{\mathsf{L}}^{*}\ \ \mathsf{P}\ \ \dot{\mathsf{e}}'' \\ \mathsf{P}\ \ \Gamma \vdash' \dot{\mathsf{e}}'' : \mathsf{t} \to \mathsf{T}' \to \mathsf{T}''\end{array}}{\mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{e}}\ \dot{\mathsf{e}}' : \mathsf{T}''} \qquad (\textsc{TyExoApp})$$

Figure 7.1: Typing rules, $\mathsf{P}\ \ \Gamma \vdash \dot{\mathsf{e}} : \mathsf{T}$, for ExoTerra expressions. $\vdash'$ refers to the same rules but with TYEXOAPP removed.

their property functions is already constructed. This environment would normally be created in Lua by API calls that create types and set entries in their metamethods table.

We model the Terra language with exotypes (ExoTerra) as the typed lambda calculus:

$$\begin{array}{rcl}\dot{\mathsf{e}} & ::= & \lambda\dot{\mathsf{x}} : \mathsf{T}.\dot{\mathsf{e}} \mid \dot{\mathsf{x}} \mid \dot{\mathsf{e}}\ \dot{\mathsf{e}} \mid \mathsf{c}_{\mathsf{Int}} \mid \mathsf{c}_{\mathsf{t}}\ \dot{\mathsf{e}} \mid \mathsf{unwrap}\ \dot{\mathsf{e}} \\ \mathsf{T} & ::= & \mathsf{Int} \mid \mathsf{T} \to \mathsf{T} \mid \mathsf{t} \\ \dot{\mathsf{v}} & ::= & \mathsf{c}_{\mathsf{Int}} \mid \lambda\dot{\mathsf{x}} : \mathsf{T}.\dot{\mathsf{e}} \mid \mathsf{c}_{\mathsf{t}}\ \dot{\mathsf{v}}\end{array}$$

Types are either a base type `Int`, a function $\mathsf{T} \to \mathsf{T}$, or an exotype represented by a type identifier `t`. An exotype constructor ($\mathsf{c}_{\mathsf{t}}\ \dot{\mathsf{e}}$) takes an expression $\dot{\mathsf{e}}$ to initialize the internal value of the exotype, and `unwrap` $\dot{\mathsf{e}}$ retrieves the internal value. The environment `P` holds a mapping from an exotype `t` to its *property lookup function*, $\mathsf{P}(\mathsf{t}) = \lambda\mathsf{x}.\mathsf{e}$. In real Terra, each property is a different function, but this is only for convenience. In our formalism, the kind of property is passed as an argument. The property lookup functions are written in ExoLua,

$$\dot{C} \quad ::= \quad \bullet \mid \dot{C}\ \dot{\mathsf{e}} \mid \dot{\mathsf{v}}\ \dot{C} \mid \mathsf{c_t}\ \dot{C} \mid \mathsf{unwrap}\ \dot{C}$$

$$\frac{\mathsf{P}\ \dot{\mathsf{e}} \longrightarrow_{\mathsf{T}} \mathsf{P}\ \dot{\mathsf{e}}'}{\mathsf{P}\ \dot{C}[\dot{\mathsf{e}}] \longrightarrow_{\mathsf{T}} \mathsf{P}\ \dot{C}[\dot{\mathsf{e}}']}\ \text{(TCtx)} \qquad \mathsf{P}\ (\lambda\dot{\mathsf{x}} : \mathsf{T}.\dot{\mathsf{e}})\ \dot{\mathsf{v}} \longrightarrow_{\mathsf{T}} \mathsf{P}\ \dot{\mathsf{e}}[\dot{\mathsf{v}}/\dot{\mathsf{x}}]\ \text{(TApp)}$$

$$\mathsf{P}\ \mathsf{unwrap}\ \mathsf{c_t}\ \dot{\mathsf{v}} \longrightarrow_{\mathsf{T}} \mathsf{P}\ \dot{\mathsf{v}}\ \text{(TUnwrap)}$$

$$\frac{\mathsf{P}\ \emptyset \vdash \dot{\mathsf{v}}' : \mathsf{T} \qquad \mathsf{runprop}\ \emptyset\ \mathsf{prop\ t\ apply\ T} \longrightarrow_{\mathsf{L}}^{*} \mathsf{P}\ \dot{\mathsf{e}}}{\mathsf{P}\ (\mathsf{c_t}\ \dot{\mathsf{v}})\ \dot{\mathsf{v}}' \longrightarrow_{\mathsf{T}} \mathsf{P}\ \dot{\mathsf{e}}\ (\mathsf{c_t}\ \dot{\mathsf{v}})\ \dot{\mathsf{v}}'} \qquad \text{(TExoApp)}$$

Figure 7.2: Rules, $\mathsf{P}\ \dot{\mathsf{e}} \longrightarrow_{\mathsf{T}} \mathsf{P}\ \dot{\mathsf{e}}'$, for evaluating ExoTerra expressions.

$$\begin{aligned} C \quad &::= \quad \bullet \mid C\ \mathsf{e} \mid \mathsf{v}\ C \mid \mathsf{prop}\ \mathsf{v}_0 ... \mathsf{v}_{i-1}\ C\ \mathsf{e}_{i+1} ... \mathsf{e}_n \mid \mathsf{runprop\ S}\ C \\ C' \quad &::= \quad \bullet \mid C'\ \mathsf{e} \mid \mathsf{v}\ C' \mid \mathsf{prop}\ \mathsf{v}_0 ... \mathsf{v}_{i-1}\ C'\ \mathsf{e}_{i+1} ... \mathsf{e}_n \end{aligned}$$

$$\frac{\mathsf{P}\ \mathsf{e} \xrightarrow{L} \mathsf{P}\ \mathsf{e}'}{\mathsf{P}\ C[\mathsf{e}] \xrightarrow{L} \mathsf{P}\ C[\mathsf{e}']}\ \text{(LCtx)} \qquad \frac{C \neq \bullet}{\mathsf{P}\ C[\mathsf{error}] \xrightarrow{L} \mathsf{P\ error}}\ \text{(LError)} \qquad \mathsf{P}\ (\lambda\mathsf{x.e})\ \mathsf{v} \xrightarrow{L} \mathsf{P}\ \mathsf{e}[\mathsf{v}/\mathsf{x}]\ \text{(LApp)}$$

$$\mathsf{P\ runprop\ S\ v} \xrightarrow{L} \mathsf{P\ v}\ \text{(LRunProp)}$$

$$\frac{(\mathsf{v}_0, ..., \mathsf{v}_n) \in \mathsf{S}}{\mathsf{P\ runprop\ S}\ C'[\mathsf{prop}\ \mathsf{v}_0 ... \mathsf{v}_n] \xrightarrow{L} \mathsf{P\ error}} \qquad \text{(LPropCycle)}$$

$$\frac{(\mathsf{t}, \mathsf{v}_1, ..., \mathsf{v}_n) \notin \mathsf{S} \qquad \mathsf{P}(\mathsf{t}) = \lambda\mathsf{x.e}}{\begin{array}{c}\mathsf{P\ runprop\ S}\ C'[\mathsf{prop\ t}\ \mathsf{v}_1 ... \mathsf{v}_n] \xrightarrow{L} \\ \mathsf{P\ runprop\ S}\ C'[\mathsf{runprop}\ (\mathsf{S} \cup \{(\mathsf{t}, \mathsf{v}_1, ..., \mathsf{v}_n)\})\ ((\lambda\mathsf{x.e})\ \mathsf{v}_1 ... \mathsf{v}_n)]\end{array}} \qquad \text{(LProp)}$$

Figure 7.3: The rules, $\mathsf{P\ e} \xrightarrow{L} \mathsf{P\ e}'$, for evaluating ExoLua.

based on the untyped lambda calculus:

$$\begin{array}{lcl} \mathsf{v} & ::= & \mathsf{T} \mid \lambda \mathsf{x.e} \mid \dot{\mathsf{e}} \\ \mathsf{e} & ::= & \mathsf{v} \mid \mathsf{e}\ \mathsf{e} \mid \mathsf{x} \mid \mathtt{error} \mid \mathsf{prop}\ \mathsf{e}_0 \ldots \mathsf{e}_n \mid \mathsf{runprop}\ \mathsf{S}\ \mathsf{e} \end{array}$$

ExoTerra types (T) and expressions ($\dot{\mathsf{e}}$) are values in ExoLua so that they can be returned from property queries. An exception expression, `error`, models the errors that occur when we find a cyclic property query. The `prop` form is used to query a property of a type. Its first argument e_0 should evaluate to an exotype identifier `t`, and $\mathsf{e}_1 \ldots \mathsf{e}_n$ are additional arguments describing the query. Properties are evaluated in the context of querying other properties. This evaluation is formalized with the `runprop S e` expression, which evaluates a property lookup expression `e` in the context `S`, where `S` is a set of properties. Each property in `S` is already being queried when `e` is requested, and has the form $(\mathsf{v}_0, \ldots, \mathsf{v}_n)$.

The rules for typechecking ExoTerra are shown in Figure 7.1. Rule TyCtor illustrates how the typechecking process queries the property function to retrieve information about the type. In particular, TyCtor asks the property function for the type of the concrete value that will represent the exotype, and makes sure it matches the type of the expression used to initialize it. Rule TyExoApp shows how a property function defines the behavior of an exotype when it is applied like a function. It queries the exotype for its behavior when applied to a value of type T' ($\mathtt{runprop}\ \emptyset\ \mathtt{prop}\ \lambda\mathtt{x.e}\ \mathtt{apply}\ \mathsf{T}'$). This query should produce an implementation of behavior in the form of a function ($\dot{\mathsf{e}}''$). This function takes the exotype $\dot{\mathsf{e}}$ and the argument $\dot{\mathsf{e}}'$ to compute the value of the expression. We check $\dot{\mathsf{e}}''$ with modified typing rules $\vdash'$ which omit TyExoApp. This change prevents the implementation $\dot{\mathsf{e}}''$ of an exotype from relying on exotype behavior itself, which can prevent typechecking from terminating if, for instance, a query about exotype application included the same application in its implementation. This behavior does not restrict what implementations can be expressed, since the property generating the implementation function can query exotypes for the appropriate behavior. Rules for evaluating ExoTerra are presented in Figure 7.2 and show how the results of property queries will be applied to evaluate Terra code.

The rules for evaluating ExoLua are shown in Figure 7.3. Exceptions abort the computation (LError). The rules LProp and LPropCycle perform property queries. A property is computed as the nested application $\mathsf{v}_0 ... \mathsf{v}_n$, where v_0 is the property lookup function for type `t`. We say that the tuple $(\mathsf{t}, \mathsf{v}_1, ..., \mathsf{v}_n)$ is the property being queried. For example, the expression `P(t) apply T` is a query of the `(t, apply, T)` property. Property statements (`prop`) can only step inside of a `runprop` rule which specifies the set `S` of active property lookups. If the same property is already being queried, it is in set `S` and will evaluate to `error` (rule LPROPCYCLE), otherwise the property will be evaluated in a new `runprop` context that records the fact that the property is currently being queried (rule LPROP). The values in property queries can be functions; for the purposes of `S`, we consider two lambda terms equal if they are equivalent up to alpha conversion. Given this formalization, we can define sufficient conditions to ensure that a property lookup during typechecking will terminate:

- *Individual termination.* A property evaluation `e` in a program $\mathsf{P}\ \ C[\mathsf{runprop\ S\ e}]$ reduces to $\mathsf{P}\ \ C[\mathsf{v}]$ or `P  error` assuming that all of the subsequent property evaluations that it evaluates ($\mathsf{P}\ \ C[\mathsf{runprop\ S}\ C'[\mathsf{prop\ v}'_0, ...\mathsf{v}'_n]]$) also reduce to values ($\mathsf{P}\ \ C[\mathsf{runprop\ S}\ C'[\mathsf{v}']]$) or `P  error`.

- *Closed universe.* There exist a finite number N of unique properties of the form $(\mathsf{v}_0, ..., \mathsf{v}_m)$ that can be queried.

THEOREM Assuming *individual termination* and *closed universe*, a property lookup $\mathsf{runprop}\ \emptyset\ \mathsf{e}$ will terminate with a value `v` or `error`.

The proof uses the fact that there is a bounded set of properties to show that a program will eventually terminate or reach a cycle.

LEMMA Let E_n be a property evaluation with individual termination, $\mathsf{P}\ \ C[\mathsf{runprop\ S}_n\ \mathsf{e}]$, that does not terminate, and $|\mathsf{S}_n| = n$. Then E_n reduces to a property lookup $E_n \ \longrightarrow^*_{\mathsf{L}}\ E_{n+1}$, with $E_{n+1} = \mathsf{P}\ \ C[\mathsf{runprop\ S}_n\ C'[\mathsf{runprop\ S}_{n+1}\ \mathsf{e}]]$ and $|\mathsf{S}_{n+1}| = n + 1$. Proof: from *individual termination*, there must exist a sequence of steps $E_n \ \longrightarrow^*_{\mathsf{L}}\ E_{n+1}$, where $E_{n+1} = \mathsf{P}\ \ C[\mathsf{runprop\ S}_n\ C'[\mathsf{prop\ v}_0 ... \mathsf{v}_m]]$ and E_{n+1} does not terminate. Furthermore, the only rule that applies to E_{n+1} is LPROP, since LPROPCYCLE terminates with an error. Hence, $E_{n+1} \ \xrightarrow{L}\ \mathsf{P}\ \ C[\mathsf{runprop\ S}_n\ C'[\mathsf{runprop\ S}_{n+1}\ \lambda \mathsf{x.e\ v}_1 ... \mathsf{v}_m]]$, where $\mathsf{S}_{n+1} = \mathsf{S}_n \cup \{(\mathsf{v}_0, ..., \mathsf{v}_m)\}$. From LPROP we know that the new property was not already in S_n, so $|\mathsf{S}_{n+1}| = n + 1$.

PROOF OF THEOREM Assume a property lookup `P runprop` $\emptyset$ `e` does not terminate. By induction using the Lemma, evaluation will step to a property lookup E_{N+1} with $|\mathsf{S}_{N+1}| = N + 1$ active properties. However, this contradicts the closed universe assumption, since there are at most N properties that can be queried.

Removing either of these conditions allows properties to run forever. If we remove *individual termination* then it is possible that an individual property lookup function will not terminate. We expect programmers can debug issues that arise from non-termination within a single property.

If we remove *closed universe*, it also possible to run forever. Consider a `__getmethod` property that, for each method `m`, appends the string `"foo"` to `m` and tries to call this new method on itself. This property will not terminate because the property being requested at each depth is different from the previous properties. In our implementation, we track which properties are being queried and throw an exception when a cycle is found. However, in cases such as `__getmethod`, there are an unbounded number of possible method names, so some property lookups may not terminate. In practice, we cap the depth of property lookup and report a trace of property requests when the limit is reached to ensure termination.

The semantics of property lookup suggest a few design principles to ensure property functions are composable. First, though Lua is not a purely functional language, property lookup functions should be written in a functional style. The semantics show that in some cases such as TYCTOR and TYUNWRAP, the same property will be evaluated multiple times. Furthermore, typechecking occurs when a function is first used, so the order in which type properties are evaluated is determined dynamically. Since the formal languages are functional, they will always produce the same result regardless of when they are evaluated. In our actual implementation where side effects are possible, we memoize property queries to guarantee the same result. Since the writer of a property does not control when it is queried, it is a good idea to write property functions so that they will produce the same result regardless of when they are evaluated.

Furthermore, it is important to avoid creating cyclic property queries. It is sometimes convenient to calculate a group of properties (e.g. all methods of a type) at once during a property lookup. This approach is problematic, since querying additional properties can cause additional cycles. Consider an analogous case when typechecking the exotype constructor $\mathtt{P}\ \mathtt{c}_{\mathtt{t}_1} 1$. Using the following property lookup functions will request an additional property causing a cycle:

$$\begin{aligned}
\mathtt{P}(\mathtt{t}_1) &= \lambda \mathtt{x}_{\mathtt{name}}.\mathtt{prop}\ \mathtt{t}_2\ \mathtt{x}_{\mathtt{name}} \\
\mathtt{P}(\mathtt{t}_2) &= \lambda \mathtt{x}_{\mathtt{name}}.\mathtt{first}\ \mathtt{Int}\ (\mathtt{prop}\ \mathtt{t}_1\ \mathtt{x}_{\mathtt{name}}) \\
& \textit{where}\ \mathtt{first} = \lambda \mathtt{x}_1.\lambda \mathtt{x}_2.\mathtt{x}_1
\end{aligned}$$

$\mathtt{t}_1$ will forward its property query to $\mathtt{t}_2$. $\mathtt{t}_2$ will first query the same property on $\mathtt{t}_1$, discard the result, and return `Int`. This evaluation would cause a cycle on $(\mathtt{t}_1, \mathtt{layout})$ that could be avoided if $\mathtt{t}_2$ only queried properties it needed. This example suggests that a property should only query other properties when they are needed to calculate the result of the original query. Querying other properties only when needed and writing properties in a functional style ensures that property queries are as composable as possible.

CHAPTER 8
BUILDING LIBRARIES WITH EXOTYPES

To evaluate the expressiveness and performance of libraries built with exotypes, we have implemented example solutions for several domains where performance is critical. In each scenario, we show how exotypes can express a solution similar to those written in dynamic languages with meta-object protocols while matching the performance of existing state-of-the-art implementations written in C++ or Java. In some cases, the added expressiveness enables more aggressive optimizations, allowing our implementations to exceed the performance of existing libraries.

Evaluation was performed on an Intel Core i7-3720QM with 16GB of RAM running OS X 10.8.5. Our implementation of exotypes was built by modifying the original Terra typechecker to make user-defined property queries while tracking cyclic property lookups. Lua's protected call mechanism was used to recover from and report any errors in user-defined properties.

8.1 CLASS SYSTEMS

Using exotypes we can implement class systems as a library. This approach is frequently used in dynamically-typed languages such as Lua that do not have built-in policies for inheritance or other familiar object-oriented features.

Using exotypes, we implement a single-inheritance class system with multiple subtyping of interfaces similar to Java's. We specify classes using an interface implemented in Lua. An example that uses this interface is shown in Figure 8.1. The function `interface` creates a new interface given a table of method names and types. The functions `J.extends` and

```
J = terralib.require("lib/javalike")
Drawable = J.interface { draw = {} -> {} }
struct Square { length : int; }
J.extends(Square,Shape)
J.implements(Square,Drawable)
terra Square:draw() : {} ... end
```

Figure 8.1: Example code that uses our class system interface

`J.implements` install property lookup functions on the `Square` type that will implement the behavior of the class system.

Our implementation inside the types themselves is based on vtables, and uses the subset of Stroustrup's multiple inheritance [69] that is needed to implement single inheritance with multiple interfaces. For each class, we define a `__getentries` property function, allowing it to compute the layout of the type. Because this property lookup function is called only when the actual layout is needed during typechecking, the calls to `J.extends` and `J.implements` will have already been performed, so all the information about the type will be known. For our class system, `__getentries` is responsible for calculating the concrete layout of the class, creating the class's vtable, and creating vtables for any interface that the class implements. If the user specified a parent class using `J.extends`, then the class and its vtables are organized such that the beginning of each object has the same layout as an object of the parent, making it safe to cast a pointer to the class to a pointer to the parent. If the user specified an interface using `J.implements` then we create a vtable that implements the interface, and insert a pointer to the vtable in the layout of the class. Finally, for each method defined in the type's `methods` table, we create a stub method to invoke the real method through the class's vtable:

```
class.metamethods.__getmethod = function(self,methodname)
    local fn = self.methods[methodname]
    local fntype = fn:gettype()
    local params = fntype.parameters:map(symbol)
    local self = params[1]
    return terra([params]) : fntype.returntype
        return self.__vtable.[methodname]([params])
    end
end
```

At this point, child classes can access the methods and members of a parent class, but the Terra compiler will not allow the conversion from a child to its parent or to an interface. To enable conversions, we create a user-defined conversion that reflects the subtyping relations of our class system (e.g., `&Square <: &Shape`). We implement the conversion generically by defining a `__cast` property function:

```
class.metamethods.__cast = function(from,to,exp)
  if from:ispointer() and to:ispointer() then
    if issubclass(from.type,to.type) then
      return `[to](exp) --cast expression to ‘to’ type
    elseif implementsinterface(from.type,to.type) then
      local imd = interfacemetadata[to.type]
      return `&exp.[imd.name] --extract subobject
  end end
  error("not a subtype")
end
```

Since the beginning of a child class has the same layout as its parent, we can convert a child into a parent by simply casting the object's pointer to the parent's type (`[to](exp)`). Converting an object to one of its interfaces requires selecting the subobject that holds the pointer to the interface's vtable (`&exp.[imd.name]`). The stubs generated for the interface restore the object's pointer to the original object before invoking the concrete method implementation.

The implementation requires only 250 lines of Terra code to provide much of the functionality of Java's class system and is comparable in size to a class system wrapper that might be implemented in Javascript or Lua. We measured the overhead of function invocation in our implementation using a micro-benchmark and found it performed within 1% of analogous C++ code. Users are not limited to using any particular class system or implementation. For instance, we have also implemented a system that implements interfaces using fat pointers that store both the object pointer and vtable together.

8.2 Data layout

Exotypes can also help programmers create container objects that can easily adapt their data layout to different use cases. One common problem in high-performance computing is choosing between storing records as an array of structs (AoS, all fields of a record stored contiguously), or as a struct of arrays (SoA, individual fields stored contiguously). We implement a solution to this problem, and contrast it with existing languages.

Changing the layout can substantially improve performance. We implemented two micro-benchmarks based on mesh processing. Each vertex of the mesh stores its position,

Benchmark	Array-of-Structs	Struct-of-Arrays
Calc. vertex normals	3.42 GB/s	2.20 GB/s
Translate positions	9.90 GB/s	14.2 GB/s

Figure 8.2: Performance of mesh transformations using different data layouts.

and the vector normal to the surface at that position. The first benchmark calculates the vector normal as the average normal of the faces incident to the vertex. The second simply performs a translation on the position of every vertex. Figure 8.2 shows the performance using both AoS and SoA form. Calculating vertex normals is 55% faster using AoS form. For each triangle in the mesh, positions of its vertices are gathered, and the normals are updated. Since this access is sparse, there is little temporal locality in vertex access. AoS form performs better in this case since it exploits spatial locality of the vertex data — all elements of the vertex are accessed together. In contrast, translating vertex positions is 43% faster using SoA form. In this case, the vertices are accessed sequentially, but the normals are not needed. In AoS form these normals share the same cache-lines as the positions, and memory bandwidth is wasted loading them.

To facilitate the process of choosing a data layout in Terra, we implemented a function that can generate either version, but presents the same interface. A Lua function `DataTable` takes a Lua table specifying the fields of the record and how to store them (AoS or SoA), returning a new Terra type. For example, a fluid simulation might store several fields in a cell:

```
FluidData = DataTable({ vx = float, vy = float,
            pressure = float, density = float },"AoS")
```

The `FluidData` type provides methods to access a row (e.g., `fd:row(i)`). Each row can access its fields (e.g., `r:setx(1.f)`, `r:x()`). The interface abstracts the layout of the data, so it can be changed just by replacing `"AoS"` with `"SoA"`.

This behavior can be emulated ahead-of-time in low-level languages, for example using X-Macros [53] in C, or template meta-programming in C++, but unlike Terra cannot be generated dynamically based on runtime feedback. Dynamic languages such as Javascript support this ad hoc creation of data types dynamically but do not provide the same low-level of control.

8.3 Serialization

Fast serialization is necessary for implementing high-performance distributed systems. Writing robust and efficient serialization libraries is difficult because different use-cases often demand different features, impacting the set of implementations and optimizations that can be used. Design choices include binary vs. text encodings, the presence of type and versioning annotations, whether complete object graphs need to be serialized, and many other considerations.

One solution is to provide robust libraries that attempt to address every possibility. Google's Protocol Buffers provide cross-language data-description and versioning [34]. Java includes a built-in library for serialization that can serialize entire object graphs and ensure type safety [42]. To optimize performance, some libraries such as the Kryo library for Java generate specialized serialization code for each type ahead-of-time [70]. The robust features in these libraries often make them difficult to customize, which is unfortunate, since advanced features such as supporting object graphs can incur runtime overhead.

Using exotypes, we can create custom serialization libraries that are generic (work on arbitrary types) and efficient (code to serialize each object is precompiled) with very little code. We focus on the example of serializing 3D scene data from an interactive scene editor connected to a storage server. Scenes are trees, so the serializer must recursively serialize sub-trees but need not handle generic graphs. Tree nodes contain a mixture of numeric and string data. The server and client exchange a binary representation of scene data.

We created a generic exotype suited to serialize this kind of data. It responds to two polymorphic methods `write` and `read`. A partial list of the code for the implementation of `write` is shown in Figure 8.3, focusing on serialization of structs (other exotypes). Code is generated for each type seen in the object tree. For `struct` objects, it queries the layout of the `struct` to generate code for each of the entries.

Performance for a 1.28MB scene is shown in Figure 8.4. We compare against Java's native serialization, the state-of-the-art Kryo library, and Google's C++ implementation of protocol buffers. Objects were serialized to a pre-allocated buffer in memory so that the benchmark would capture encoding time rather than buffer allocation/resizing time. For Java implementations, the JIT was warmed up by serializing the entire scene 250 times

```
createwrite = memoize(function(T)
  if T:isstruct() then
    local function emitPointers(self,obj)
      local stmts = {}
      local function addEntry(elemtype,elemexp)
        if elemtype:ispointer() then
          local fn = createwrite(elemtype.type)
          table.insert(stmts,quote fn(self,elemexp) end)
        elseif elemtype:isstruct() then
          local entries = elemtype:getentries()
          for _,e in ipairs(entries) do
            addEntry(e.type,`elemexp.[e.field])
      end end end
      addEntry(T,obj)
      return stmts
    end
    local terra write(self : &Serializer, obj : &T)
      self:writebytes([&uint8](obj),sizeof(T))
      [emitPointers(self,obj)]
    end
    return write
  elseif T:isarray() then ...)
```

Figure 8.3: Implementation of a generic serializer, focusing on code for serializing aggregate types.

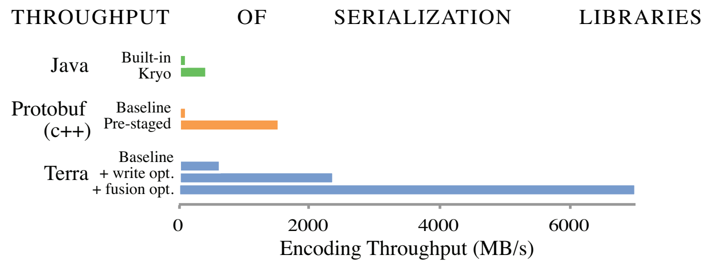


Figure 8.4: Throughput of scene graph serialization. Our specialized exotype implementation performs an order of magnitude faster than existing implementations.

before timing. For Kryo, classes were pre-registered, support for object graphs was disabled, and `UnsafeMemoryOutput` was used to increase performance.

Our baseline implementation can serialize input data at 627 MB/s, which is comparable to Kryo's 432 MB/s. This is expected since both libraries take the same approach, pregenerating code to serialize each object up front. The Java serializer performs substantially worse (107MB/s) since it interprets the structure of each object during serialization.

Given a specific use-case, we can apply more aggressive optimizations. Our baseline implementation calls a user-provided function pointer to write data. We can turn our serialization type into a type-constructor that takes the write function as an argument. This change (write opt.) allows inlining calls to the write function, increasing performance to 2.37 GB/s. Our baseline implementation also writes each element of a struct individually,

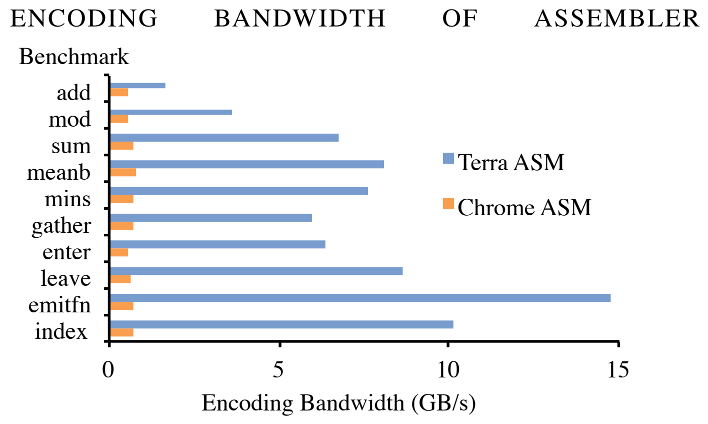


Figure 8.5: Throughput of assembler. Benchmarks are ordered by increasing template size. Larger template sizes increase the effect of the Terra assembler's fusion optimization.

allowing for customized writers for specific objects. However, in this example, no custom writers are needed, so the library can apply aggressive fusion to the writes. Rather than copy each struct element individually, it copies the entire struct at once. Any objects pointed to by elements of the struct are then written afterward. Using vectorization, these larger copies are efficient, increasing the performance to 7.00GB/s (fusion opt.) or 11 times faster than Kryo. As an upper bound on performance, simply copying 1.28MB runs at 15GB/s.

Protocol buffers use a different approach, requiring the user to translate an object into its own object hierarchy before serialization. This translation step limits performance to 124MB/s. Serializing the protocol buffer objects directly runs at 1.5GB/s (pre-staged), but doing so is not always possible since it prevents the programmer from using their own object hierarchies in the rest of the program.

The combination of staged programming and meta-object protocol makes the expression of this custom serializer concise. In the full Terra implementation, the code to implement both serialization and deserialization is less than 200 lines. At this size, it is feasible for a programmer to modify it to fit different serialization cases, something that is not as feasible with much larger serialization libraries such as Kryo or protocol buffers which each contain several thousand lines of code.

8.4 Dynamic x86 assembly

Dynamic assemblers form the basis of JIT compilers. Unlike normal assemblers which run during compilation, dynamic assemblers are used at runtime to dynamically generate code. A JIT compiler uses the assembler to translate its own intermediates into executable machine code. Higher-level JIT-compiled languages may emit *templates* of x86 assembly that correspond to a single high-level instruction. For instance the Riposte JIT [74] for vector code in R uses the following template to emit a vector gather:

```
void emitGather(Assembler &a,
                XMMRegister RegR, XMMRegister RegA, int disp) {
    a.movq(r8, RegA); a.movhlps(RegR, RegA); a.movq(r9, RegR);
    a.movlpd(RegR,Operand(r12,r8,times_8,disp));
    a.movhpd(RegR,Operand(r12,r9,times_8,disp));
}
```

The gather takes two addresses in `RegA` and then loads the two values in `RegR`, requiring five instructions total.

The performance of a JIT depends on compilation speed, so it is important for the `Assembler` type to be efficient. Riposte uses the assembler used in Google Chrome [1] to get high performance. To ensure speed, each instruction is explicitly implemented as a method on the assembler object. As an example, here is the implementation of the `movlpd` instruction:

```
void Assembler::movlpd(XMMRegister dst,
                       const Operand& src) {
  EnsureSpace ensure_space(this);
  emit(0x66); emit_rex_64(dst, src);
  emit(0x0F); emit(0x12); // load
  emit_sse_operand(dst, src);
}
```

This approach is able to produce code that can assemble x86 instructions at the rate of 720 MB/s of instructions. While fast, writing the implementation of each of these functions is tedious and prone to error. There are hundreds of x86 instructions, many with multiple versions.

Another approach is to describe the instruction concisely in a small language, similar to how string matching can be encoded with regular expressions. LuaJIT's DynAsm library [2] takes this approach to describe the `movlpd` instruction in a table:

```
movlpd_2 = "rx/oq:660F12rM|xr/qo:n660F13Rm"
```

Each line describes the valid arguments (`"rx"`), their sizes (`"oq"`), and a recipe to encode the instruction (`":66..."`), listing multiple variants per line. When instructions share a similar form (e.g., `add` and `sub`), meta-programming is used to generate the table entries.

While these tables are concise, interpreting the table to encode instructions incurs substantial overhead. A micro-benchmark using this table directly encodes at only 336KB/s, three orders of magnitude slower. To get high-performance, DynAsm pre-compiles the table into fast code using a source-to-source translation of C code. It is designed to optimize code size rather than speed, encoding the gather code at 168MB/s but using only 30 bytes to represent it.

We can use exotypes to perform this transformation directly in the object system of Terra without the need for preprocessors. Instead of optimizing for code size like DynAsm, we optimize for encoding speed. Our implementation uses the `__methodmissing` property to compile assembly functions on demand. For instance, a user may write:

```
A:movlpd(RegR,index(r12,r8,8,disp,"qword"))
```

This will invoke `__methodmissing` for the `movlpd` instruction. The implementation will then examine the encoding table (adapted from DynAsm) to produce an implementation of the instruction equivalent to C code shown previously from the Chrome assembler.

Generating the assembler implementations on demand provides more opportunities for optimization. Only instructions that are actually used need to be generated, reducing the total amount of code in the library. Also, we can aggressively specialize instructions to their use. For instance, in the invocation of `movlpd` above, the only dynamically determined arguments are `RegR` and `disp`. We can generate a specific version of `movlpd` for this invocation.

To generate specialized assembly instructions we take the following approach similar to DynAsm. First, in `__methodmissing`, we determine which arguments are constants and which are determined dynamically. We then use the constant arguments to create a *template* that leaves the dynamically determined information blank. Finally, we generate code that first copies the template into the code buffer, and then patches it up with dynamically provided arguments. This approach keeps the code size small by separating the template from the assembler code. Furthermore, since the template is normally multiple bytes, we can benefit from vectorized copies. We can also use this template-based approach to generate assembler code for *multiple instructions* in a single template. The programmer can call `emit`, which supports multiple instructions. The gather operator expressed using this approach is shown in Figure 8.6. This function fuses the assembly into a single template copy followed by

```
terra emitGather(RegR : 0, RegA : 0, disp : int)
    A:emit(op.movd, r8, RegA,
    op.movhlps,RegR, RegA,
    op.movd,r9, RegR,
    op.movlpd,RegR,index(r12,r8,8,disp,"qword"),
    op.movhpd,RegR,index(r12,r9,8,disp,"qword"))
end
```

Figure 8.6: Using our dynamic assembler, we can emit multiple instructions in a single template.

patch-up instructions to insert `RegR`, `RegA`, and `disp`. The resulting code runs at 5.96 GB/s, or 8.3 times faster than the hand-written Chrome version.

To validate the approach, we rewrote 10 kernels taken from the Riposte JIT compiler using our dynamic assembler written in Terra and compare their performance to those of the original code which uses the Chrome assembler. Figure 8.5 shows the results for each kernel. While the Chrome assembler always emits code at a rate of 700MB/s, the Terra assembler can perform anywhere from 1.64 GB/s to 15 GB/s, depending on the amount of instruction fusion that is possible. For instance, the `add` instruction only includes two x86 instructions, and one of them is only emitted conditionally. This limits performance to 1.64 GB/s. The `emitfn` kernel emits 13 instructions and does little patching, enabling it to encode at 15GB/s.

Using exotypes we were able to implement our Terra-based assembler, including the parts of DynAsm we used to implement instruction encoding, with only 2100 lines of Lua-Terra code—less than half the size of the Chrome assembler. Furthermore, by specializing the assembler code for each call to the assembly object, we were able to produce assembly code that ran up to 17 times faster than the reference code.

8.5 Automatic differentiation

Automatic differentiation (AD) computes derivatives of programs by differentiating elementary operations (such as multiplication) and composing those derivatives using the chain rule [19]. It eliminates tedious and error-prone hand-authoring of derivative code, and it gives exact derivatives, unlike finite difference approaches. AD is widely applied to sensitivity analysis and optimization in scientific computing and engineering.

Many applications, such as optimizing an objective function, require the gradient of a single output with respect to multiple parameters, a setting well-suited to *reverse-mode* AD.

Reverse-mode AD first runs the program forward, recording each elementary operation and the data needed to compute its derivative on a *tape*. It then interprets the tape backward, accumulating the partial derivative of the output with respect to each intermediate (the intermediate's *adjoint*), terminating with the partial derivatives for the input parameters. Reverse-mode AD can compute arbitrarily many partial derivatives with just two sweeps through the program (one forward and one backward), but the space overhead for the intermediate tape may be significant.

We implemented a reverse-mode AD library with exotypes in Terra, using an approach similar to that of Stan, a C++ library for high-performance statistical inference [67]. Programs written against this library replace floating point numbers with a custom *dual number* type for which arithmetic functions and operators are overloaded. These overloaded functions store their inputs in an object which is placed on an in-memory tape. The reverse pass interprets the tape by calling virtual functions on those objects.

Our implementation uses exotypes to programmatically generate the tape object type for each elementary operator. New operators are defined using a simple interface:

```
-- Defining the "__add" metamethod
addADOperator("__add",
-- Forward function code
terra(lhs: double, rhs: double) return lhs + rhs end,
-- Adjoint code
adjoint(function(T1, T2)
return terra(v: &TapeObjBase, lhs: T1, rhs: T2)
    setadj(lhs, adj(lhs) + adj(v))
    setadj(rhs, adj(rhs) + adj(v))
end end))
```

`v` is the output of the add operator stored on the tape. The inputs to the operator, `lhs` and `rhs`, may be either doubles or dual numbers. `adj(x)` extracts the adjoint of `x`, and `setadj(x, v)` sets the adjoint of `x` to `v`. In the above example, when `lhs` or `rhs` is a double (i.e., a program constant), the first `setadj` line is unnecessary, since `a` has no adjoint. Our implementation detects this at compile time (`adj` and `setadj` are macros) and does not add an entry for `lhs` to the tape object type. In contrast, Stan uses a class hierarchy for tape objects: `Add` is a subclass of `BinaryOp`, whose subclasses all have the same layout. Our approach helps alleviate the memory overhead that is the main drawback of reverse mode AD.

We evaluate the runtime and memory performance of Terra AD and Stan C++ AD on a standard optimization task from machine learning: maximum-likelihood training of a

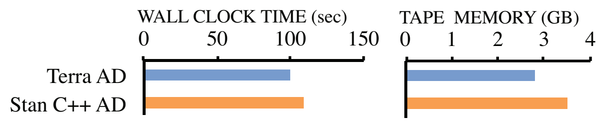


Figure 8.7: Runtime performance and peak memory usage for reverse-mode AD in Terra and C++. Our Terra implementation achieves comparable speeds and a 25% smaller memory footprint.

logistic regression classifier for hand-written digits using the MNIST dataset [50]. For each implementation, we calculate the gradient of data log probability with respect to model parameters using 6000 data points, and we run 100 iterations of gradient descent. Results are shown in Figure 8.7. The Terra code achieves runtime performance comparable to C++ with 25% less peak tape memory usage.

Our Terra AD implementation takes 493 lines of code, compared to Stan's 1187 lines. This difference is due to programmatic type generation, instead of explicitly-defined class hierarchies. While the core of each library (i.e., tape management and public interface) takes roughly the same amount of code (260 vs. 318 lines), adding new elementary operators is more concise in Terra ($\sim$10 vs $\sim$60 lines for a new binary operator).

Libraries like AD can be easily integrated into existing Terra applications that use exotypes. The dual number type used in AD, for instance, can be used in the implementation of Quicksand's tracing infrastructure to automatically compute derivatives of program probabilities with respect to random choices. We have implemented an MCMC algorithm that uses these derivatives to perform more efficient inference for programs defining mostly-continuous distributions [54].

CHAPTER 9
EMBEDDING CUSTOM DSLS IN LUA

The two-language embedding of Terra and Lua is designed to make generating low-level code easy. While aspects of this design are specific to the problem of embedding a system programming language in a high-level language, many other languages can benefit from adapting some of its designs, such as using shared lexical scoping for meta-programming during specialization, and having a separate execution pathway for high performance. This chapter presents our extension to the Lua language that allows programmers to embed other languages inside Lua using the same principles that we used to embed Terra.

9.1 CURRENT APPROACHES TO EMBEDDING LANGUAGES

All DSLs have a frontend that allows users to write statements in the language. Some examples of common approaches are shown in Figure 9.1. The most direct approach is to write a custom parser that reads a stream from some data source. Both SQL and OpenGL/Direct3D shading languages work this way. They are distributed as libraries that provide API functions that load a string representing the program.

Custom parsers provide the most flexibility for DSL syntax but make it harder for the DSL to interact with the program it is embedded in. Another approach is to use calls to an API to construct a representation of a program incrementally. This is the approach used by OpenGL's immediate mode that is used to describe geometry (Figure 9.1(c)). API calls indicate the beginning and ending of geometry such as triangles and their points are described by more function calls. While verbose in a language such as C, in languages with the ability to define custom behavior for operators (e.g., `+-/*`), these operators can be defined to create

Standalone
(AWK)

The DSL uses a custom parser and is run as its own process. Interaction is done through inter-process communication, in this case through standard input and output.

```
NR > 1 && NF == 6 { print $1 $2; }
```

Embedded as Strings
(SQL in Java)

Syntax is embedded in a string value of another language and executed via an API. Interaction is done through the API. Code for the DSL is frequently generated and run in the same process.

```
String query = "SELECT * FROM people WHERE age > 18";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

Embedded as Function Calls
(OpenGL immediates in C++)

The frontend is expressed as calls to API functions, which internally build a representation. This approach makes meta-programming easy compared to using string formatting operators to build larger expressions. Here we show how to use OpenGL to draw 100 triangles by meta-programming the display language using a loop in C++.

```
for(int i = 0; i < 100; i++) {
    glBegin(GL_TRIANGLES);
        glVertex2f(0.0f, 0.0f);
        glVertex2f(1.0f, 0.0f);
        glVertex2f(1.0f, 0.0f);
    glEnd();
    glTranslate2f(1.0f,0.0f);
}
```

Emedded using Operators
(Halide in C++)

Some languages allow user-defined operators, which can be used to make the DSL's syntax more natural, while still using an API to build the internal representation for the DSL. Note that some syntax, such as the declaration of `x` and `y` for the `blur_x` require multiple expressions in C++ since the syntax of C++ is not fully extensible.

```
Func blur_x;
Var x, y; //declare variables to be used in blur_x expression
blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
```

Figure 9.1: Examples of different techniques for implementing the frontend of a DSL

statements that build a program in the DSL. For instance in C++, Intel's Array Building Blocks enable runtime generation of vector-style code using a combination of operator overloading and macros to build a graph of vector-wide computations dynamically [56].

This approach is similar to type-directed staged programming described in Section 2.2.3. The DSL is staged by the embedding language. It shares similar advantages, namely that the DSL code itself can be meta-programmed by the host language.

Meta-programming of DSLs can be useful since it can reduce the number of built-in operations that the DSL needs to implement, leaving other operations to be defined via meta-programming. It also allows other DSLs to *generate* DSL code, allowing the programmer to layer a high-level DSL on a lower-level DSL. This approach has been used in the past for existing DSLs. Object-relational mappings frequently meta-program SQL, and shader languages such as Spark [28] generate code for traditional shading languages.

However, building the DSL frontend using type-directed operators is limited by what operators can be redefined in the embedding language. Languages such as C++ only allow limited definition of arithmetic and assignment operators, but do not, for instance, allow the redefinition of a conditional statement, or a variable declaration. Related work in the Scala community makes almost all language syntax overloadable, making a so-called "virtualized" language [12]. For instance, Chafi et al. use lightweight modular staging [62] to stage a subset of the Scala language. The staged code is used to implement DSLs in the Delite framework that can be translated to run on GPUs [7, 13].

If the embedding language and DSL are both statically-typed, then this approach also limits the types in DSL to a *subset* of those in the embedding language. This limitation can be problematic for many designs. In our experience, most DSL type systems are relatively simple, but almost always have features that would lie outside of the type system of a particular general-purpose language. For instance, Liszt's field type is a parameterized map, which is supported in many languages. But it also has limits on when it can be used, becoming read-only or write-only access in certain contexts. The rules for these contexts are simple, but fall outside the scope of most static type systems. In fact, this restriction complicated the first version of Liszt, which was embedded in Scala. It needed to perform a separate pass to check the correctness of field phases, weakening the benefits of static typing.

9.2 OUR LANGUAGE EXTENSION INTERFACE

Our solution is based on the idea that most DSL frontends face many of the same challenges that we faced when integrating Terra into the Lua ecosystem. We want other developers to be able to generate DSL code dynamically and interoperate with the DSL, but it is important that the DSL can exist as a separate language, with different syntax, semantics. and types.

We provide an API to embed a DSL in Lua using the same techniques that Terra itself uses. DSLs provide rules for parsing custom expressions embedded in the same file as normal Lua programs. Like Terra's custom expressions (e.g., `terra` and `struct`), DSL expressions result in first-class Lua values, and can contain semantics that perform DSL analogs of specialization and typechecking. This approach will allow DSLs to define their own syntax and their own custom types. By default, the DSLs also share the same lexical scope as Lua and Terra programs, making interoperability easier.

Figure 9.2 shows an example of the interface DSLs use to define frontends. Each DSL defines it own parser extension, using an API defined in Lua. Parser extensions are loaded with a special `import` statement that loads a Lua package that defines the language using the API. This package defines a set of new *keywords* that indicate a start of a DSL expression. When a keyword is encountered at the beginning of an expression or statement, the Lua parser transfers control to the DSL's custom parser. This parser is free to define its own syntax for any expression. For simplicity, we constrain the DSL parser to use the same *lexer* interface as Lua. When implementing DSLs such as Darkroom or Liszt, we have found that this balance of fixed lexing but custom parsing was flexible enough to create custom statements while simple enough to produce concise parsers. It also encourages languages to use syntax that looks consistent, since it prevents the invention of new tokens.

DSL parsers are written in Lua, using a lexer API to read tokens from the input. Internally, these parsers can use any parsing technique, and are provided one symbol of lookahead. In practice, the Lua grammar itself is amenable to top-down precedence (Pratt) parsing [59] and DSLs that are designed with a similar set of constructs are often easily parsed using this technique. For Liszt and Darkroom, we found this simple technique to be sufficiently powerful.

This example code uses an `import` statement to enable the `sumlanguage` module, which adds a new expression to Lua that sums up variables:

```
import "sumlanguage"
a,b,c = 1,2,3
result = sum
           a b c
         done
print(result) -- 6
```

The `sumlanguage` module itself is loaded by Lua's package manager, and returns a table that defines the extension behavior. The table includes information about what keywords are added to the language, and a function that handles the parsing for the extension. The parsing function itself returns a closure that will be run when the statement should be evaluated.

```
return {
    name = "sumlanguage"; --name for debugging
    entrypoints = {"sum"}; -- list of keywords that will start our expressions
    keywords = {"done"}; --list of keywords specific to this language
    -- Lua parser calls this function when it sees an expression that starts with 'sum'
    -- 'lex' is a handle to the lexer object, which has an API for producing tokens of input
    expression = function(self,lex)
        local variables = terralib.newlist()
        lex:expect("sum")
        while not lex:matches("done") then
            local name = lex:expect(lex.name).value
            variables:insert(name)
            lex:ref(name) --tell the Terra parser we will access a Lua variable, 'name'
        end
        lex:expect("done")

        -- Return the closure that will be called when this expression is actually
        -- executed in Lua
        return function(environment_function)
            --Capture the local environment, a table from variable name to value
            local env = environment_function()
            local sum = 0
            for i,varname in ipairs(variables) do
                --Capture the local environment, a table from variable name to value
                sum = sum + env[varname]
            end
            return sum
        end
    end;
}
```

Figure 9.2: An example language extension that defines a sum expression that adds variables together. This example is not an entire DSL, so its intermediate representation is just a list of variables names, but it still illustrates the different components of a DSL extension.

Like all operations in Lua and Terra, parsing runs in the same process as other stages of the execution, so a DSL can build any internal data structures while parsing and use it later in execution. Typically, a DSL will build a custom AST representing the parsed code.

When the custom DSL parser reaches the end of the expression, it transfers control back to the Lua parser. It passes a *closure* back to the Lua parser, which will be run whenever a normal Lua expression would run as if it had occurred in the same position as the custom syntax. When run, this closure will be passed a Lua table containing the local lexical environment of the program. This environment makes it possible for custom DSLs to perform a specialization phase similar to Terra's specialization. At this point DSLs are free to implement the rest of their compilation pipeline in the way that is best for the DSL. For instance, some may choose to perform typechecking lazily, like Terra, but others are free to perform typechecking eagerly if that works better for the particular DSL.

9.3 Using language extensions

To evaluate the effectiveness of this interface, we implemented two DSL frontends using this system, one for the Darkroom language, and one for the updated version of Liszt written in Terra.

9.3.1 *Darkroom: Lua-based meta-programming with custom syntax*

The original frontend for Darkroom constructed programs in Lua using operator definitions on Lua metatables. Individual image definitions existed as first-class Lua objects, and can be combined using operators such as `+` into new images expressions. Meta-programming of Darkroom could be done to generate image processing operators such as `convolve` (Figure 9.3(a)), which would take an arbitrarily sized kernel and create a convolution with an image.

This embedding worked for simple expressions such as doing arithmetic on images, but introduced some syntactic noise for other operators. Stencil expressions are normally written with explicit indexing variables for each dimension (e.g., `foo(x+1,y) + foo(x,y) + foo(x-1,y)`, with `x` and `y` representing a dimension of the image). But in Lua, `x` would need to be a Lua value, which could lead to confusion if `x` were not in scope or shadowed. In our embedding, *Lua Operators*

This example Darkroom code defines an interface for quickly generating convolutions against constant kernels. The `convolve` function generates the convolution of a Darkroom image with the kernel `K`. The * and + operators are defined on the Darkroom image type to build an IR for the image. The `convolve` function uses two loops written in Lua to construct the entire expression in Darkroom.

```
terralib.require("darkroom.operators")
-- generate a convolve function against a kernel K
-- via meta-z
local function convolve(N,K,image)
  local R = 0
  local W = 2*N+1
  for j = -N, N do
    for i = -N, N do
      local k = K[(j + N)*W + i + N + 1]
      R = R + k*image(i,j)
    end
  end
  return R
end
local blur = {0,1/5,0,
              1/5,1/5,1/5,
              0,1/5,0}
local result = convolve(3,blur,darkroom.load("input.bmp"))
```

Custom Syntax

We can also write `convolve` using Darkroom's custom embedding syntax. Here we use a map-reduce expression which is not easy to represent with operator overloading in Lua. Like expressions in Terra, Darkroom expressions share Lua's lexical scope to make meta-programming possible, illustrated by how the image expression refers to the kernel size `N`, and the kernel data `K`:

```
-- enable darkroom's syntax extensions
import "darkroom"

-- the same convolve,
-- but using Darkroom's map-reduce operator
local function convolve(N,K,image)
  local W = 2*N+1
  -- use darkrooms im operator to create a new image
  local R = im(x,y)
            map i = -N,N j = -N,N reduce(sum)
              k = K[(j + N)*W + i + N]
              k*image(x+i,y+j)
            end
          end
  return R
end
```

Figure 9.3: Embedding Darkroom in Lua using both Lua's operator overloading and custom syntax via our language extension interface.

we choose to drop the `x` and write the expression as `foo(1,0) + foo(0,0) + foo(-1,0)`. The argument to the image indicates an image-wide "shift" operation instead of an index. While semantically identical, this way of expressing the problem made the embedding different than most other languages. Furthermore, as we added new features such as conditional operators (`if x then y else c`), it became clear that forcing everything into Lua syntax would not result in ideal syntax.

Using the parser extension mechanism, we are able to create custom syntax for all operators. The expression `im(x,y) foo(x+1,y) + foo(x,y) + foo(x,-1) end` both creates a new image and concisely introduces variables for the dimensions of the image. Like Terra, Darkroom has its own specialization phase where constants from the Lua environment can be inlined into Darkroom code. In this case, it is used to resolve external references to other image functions such as `foo`. Though it allows custom syntax, this design can still generate parameterized image processing code using meta-programming. Figure 9.3(b) shows the same definition of a convolution template using the custom embedding. It uses Darkroom's *map-reduce* operator to implement the convolution, rather than unrolling the convolution entirely as was done in the original version. Since Lua does not have a built-in map-reduce operator, it would have been awkward to represent it using an approach based purely on operator overloading. But we found that representing the map-reduce in Darkroom was important for performance because it enables more efficient code generation and better code quality by capturing more structure about the computation.

The custom parser for Darkroom is also relatively simple. It is implemented in only 250 lines of code, using a helper library for top-down precedence parsing that is around 100 lines of code. Most expressions are just straightforward code that parses an expression and builds an IR.

9.3.2 *Liszt: Custom syntax with a language extension API*

The first version of the Liszt language uses Scala as an embedding language for the compiler. A compiler plugin to the Scala compiler allows Liszt to extract a typechecked version of the Liszt AST before further processing. This approach allows for easy parsing and typechecking, but runs into difficulty since it requires embedding the type system of Liszt as a subset of the embedding language. Certain features, such as constant-length vectors, and fields with read-only and reduce-only phases do not fit into Scala's type system and had to be checked at a later phase anyway. Furthermore, many operations that were not in kernels, such as the declaration of fields and sets, require their own syntax and bookkeeping. This bookkeeping occurs twice, once in the compiler to parse the declarations, and then again in the runtime to lookup the declaration and link it to existing input data (e.g., loading that data from a file).

The Lua version of Liszt simplifies many aspects of this design. Similar to how Terra exotypes can be constructed using Lua expressions, Liszt uses Lua meta-programming to create types such as fields and then load data into them. Fields and relations between mesh elements are created using a Lua API, rather than requiring custom syntax. Since the entire program executes as a single process, bookkeeping for fields and mesh relations only occurs once, and does not have to be serialized between compiler and runtime.

The Liszt kernels themselves are a full language with variable declarations, conditionals, and control flow in addition to custom statements such as reductions, and could not be implemented using a Lua API. We implemented this part as a language extension introduced by the `liszt` keyword. Similar to Darkroom, kernels can be parameterized over fields using meta-programming in Lua. Previously Liszt required all fields to be explicitly declared to make it possible to check the phases of fields without performing sophisticated alias analysis. Using Lua meta-programming, a programmer can now generate multiple versions of a function that use different fields.

The implementation of Liszt's custom parser was longer than Darkroom's at about 450 lines since it needs to parse an entire language, and includes support for Liszt-specific features such as reductions, and relational operations such as insert or delete. It is less than 10% of the codebase and a relatively minor part of the project.

9.4 Comparison to other approaches to syntax extension

Other approaches to syntax extension such as SugarJ [26], a parser extension for Java, attempt to make the process of extending the parsing more concise. SugarJ, for example, specifies new grammar rules using a declarative model based on context free grammars

(CFGs). This model has some advantages. Our extension mechanism will detect two languages as ambiguous if they both introduce statements that start with the same keyword, while SugarJ can support CFGs that start with the same keyword as long as the entire expression is not ambiguous. It also allows a declarative, rather than programmatic, expression of the new syntax. These types of extension mechanisms are valuable when trying to add a few lightweight extensions to a language such as adding tuples or XML literals demonstrated in SugarJ. In the case of larger DSLs, such as the ones we have implemented in Terra, it is actually valuable to denote the beginning of DSL code with a unique keyword so that users know what language follows. Furthermore, the parsing for both Liszt and Darkroom is less than 10% of their codebase, so we have found that efforts to make the language extensions more concise are not worth pursuing at the cost of extra complexity in the design of the extension mechanism.

CHAPTER 10
INTEROPERABILITY

High-performance code is normally written as a part of a larger application. This is especially true for DSLs and active libraries, whose programming model may be limited to specific functionality such as rendering or simulation. Any solution for making it easier to write high-performance code using strategies like DSLs needs to address how it will interoperate with other technologies. This chapter examines the design decisions we made in creating Terra and integrating it with Lua that maximize this interoperability.

10.1 WITH EXISTING APPLICATIONS

Most DSLs or active libraries are actually used as a part of a larger application. OpenGL is normally a small part of a game engine which might use other technologies to do physical simulation and artificial intelligence. If developers can replace parts of an application piecewise, they are more likely to adopt newer technologies such as DSLs. This pathway is also an advantage to DSL authors, since it means that the DSL can be responsible for running only a few parts of an application. It can grow to accelerate more of an application over time as it gains features. Terra supports this methodology at two levels. First, our use of Lua enables developers to embed the entire Lua-Terra ecosystem into any application that can call C functions. Second, Terra code and data-structures are generated such that they are compatible with C functions at the application binary interface, making them directly usable in pre-existing code.

```c
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"

const char * str =
" function add(a,b) return a + b end "
" print(add(1,1)) ";

int main(int argc, char ** argv) {
    lua_State * L = luaL_newstate(); //create a plain lua state
    luaL_openlibs(L);                //initialize its libraries
    lua_dostring(L,str);
    return 0;
}
```

Figure 10.1: Embedding Lua into a C program is relatively simple. These calls set up a Lua context `lua_State` and run the Lua code in a string.

10.1.1 *Embedding as a runtime*

We selected Lua as a high-level language because it was designed originally to be embedded in existing applications. The design uses a simple C API based on a virtual stack machine to expose the Lua environment to another application [40, 41]. The stack machine allows developers writing in C to extract primitive values from the Lua environment, as well as load and execute arbitrary code. Figure 10.1 shows example code that loads Lua as a library in a C program, and uses it to evaluate Lua code embedded in a string. These Lua APIs also make it easy to register C functions with Lua so that Lua code can call back into the surrounding application. This simple API is widely used in the game development community to couple a scripting environment into game engines. While not all applications are developed using the C runtime, almost all environments have some way of calling C functions, making this interface a least-common-denominator for interoperability.

We extend Lua's interface in our design of Terra by adding specific functions to initialize Terra and load Terra code. Figure 10.2 shows the changes necessary to an application that uses Lua to embed the Terra environment as well. Developers still use the standard Lua API to work with the Lua environment, and only need to use the Terra API when using features specific to Terra, such as evaluating a string that includes both Lua and Terra code.

Note that using Terra requires no more machinery than loading any other C library. Since it runs dynamically, the developer also gets the benefits of runtime code generation, including simplicity of design and the ability to adapt to runtime information. Requiring

```
#include "terra.h"

const char * str =
" terra add(a : int,b : int) return a + b end "
" print(add(1,1)) ";

int main(int argc, char ** argv) {
    lua_State * L = luaL_newstate(); //create a plain lua state
    luaL_openlibs(L);                //initialize its libraries
    terra_init(L); //initialize the terra state in lua
    terra_dostring(L,str);
    return 0;
}
```

Figure 10.2: Using Terra in addition to Lua only requires one additional function call to initialize the Terra state.

the developer to pre-compile Terra code (for instance, using a Makefile in their build script) would constrain certain designs relying on runtime code generation, and increase the complexity of using Terra with an existing system by forcing the developer to write custom build logic.

However, embedding Terra as a runtime library is not always the right choice. Terra includes an entire optimizing compiler (LLVM), so it is a large binary. Compile times in LLVM are also fairly slow, on the order of a few milliseconds for reasonably sized functions. This makes the runtime code generation approach less desirable for slower architectures, or where storage space is limited, such as on embedded devices. To address these uses, Terra can also produce code for offline use using the `terralib.saveobj` function which can output offline executables or shared libraries that contain pre-compiled code. In this model, the Lua environment serves as a pre-processor and compiler, producing object files that can be used apart from the Terra compilation environment. We are able to provide this functionality because Terra's semantics allow it to execute separately from the Lua environment. By offering both alternatives, Terra provides the flexibility for prototyping, with a path for optimizing compile times and binary size using selective offline compilation.

10.1.2 *Zero-overhead compatibility with C*

Embedding solves one half of the interoperability problem with existing applications – those applications can call into Lua-Terra code. It is also useful to go the other direction – have

	C	*Terra*
Primitives	Translated to built-in equivalents	

	`uint32_t uint64_t`
	`float bool`
		`int32 uint64`
		`float bool`

Pointers	`T*`	`&T`
Functions	`T3 (*)(T,T2)`	`{T,T2} -> T3`
Vectors	C equivalents use non-standard GCC extension	

	`float __attribute__`
	`((vector_size(16)))`
		`vector(float,4)`

Opaque	`void *`	`&opaque`
Arrays	True Terra arrays are passed by value, so C arrays become pointers in Terra.	
	`float[4]`	`&float`

*Structs**

```
typedef struct {
  float real;
  float imag;
} Complex;
```

```
struct Complex {
  real : float
  imag : float
}
```

Unions

In Terra, unions are a property of the layout of a struct rather than a separate type, removing the need for C's named vs. anonymous union distinction.

```
typedef union {
    float f;
    int i;
} Number;
```

```
struct Number {
    union {
        f : float;
        i : int;
    }
}
```

Other

Types such as enumerations or bit-fields are not directly translated since there is no machine equivalent to them. In Terra these types can be constructed via meta-programming. Qualifiers like `const` are ignored for simplicity since we cannot ensure their correctness given unsafe casts anyway.

* More generally, Terra structs can use the exotype API to construct their layout using the `__getentries` meta-method programmatically. The result of that function is then used to construct a corresponding C `struct`.

Figure 10.3: Correspondence between Terra types and C types. Terra types have the same memory layout as their C equivalents to ensure zero-overhead ABI compatibility.

Terra code capable of calling existing code. Furthermore, we would like compiled Terra code and data structures to be usable from existing C code without any overhead. To accomplish both goals, Terra generates functions in such a way that they are compatible with the C application binary interface, or ABI. Once a Terra function is generated, it can be linked against C functions as if they were written in the same language. The data layout of Terra types have a one-to-one correspondence with C data types, so they can be accessed directly from C without any layer of translation. Primitive data types like `uint64` map to their C equivalents (`uint64_t`), and Terra's exotypes map to C `struct`s with equivalent data layouts. Figure 10.3 describes the correspondence in detail.

This design enables several powerful behaviors. A Terra function can call a C function without any overhead since, as a part of the ABI, they use the same calling convention. We also provide a library call (e.g, `terralib.includec("stdio.h")`) that uses the Clang compiler frontend to parse C header files, and exposes all of the C functions as Terra function objects in the Lua environment. Since Clang produces LLVM similar to Terra, we can even *inline* C functions into Terra functions when their definitions are available in header files.

This interface provides a zero-overhead way to use C functions in Terra. Other languages that can call externally defined functions all require some form of declaration of each function used (e.g., Figure 10.4) that translates the language's types into equivalent C types. To our knowledge, Terra is currently the only language that does not require any declarations while still ensuring that the type of the variables passed to the C function are correct. The lack of declarations makes it feasible to write high-performance libraries in C and then use them in DSLs without making it any more difficult to write the code generator in Terra. In all cases, developers calling C libraries from Terra can expect to get the same performance as if they were calling those libraries directly from C.

When building Liszt in Terra, we were able to reuse most of the library code for loading and manipulating meshes, since it was written in C/C++. Terra can also reuse existing APIs for external libraries. In practice, we have used these features to render images with OpenGL, launch CUDA kernels on GPUs, and interface with the Objective-C runtime. Even simple tasks, such as dynamically allocating memory, are handled by calls into C's standard libraries.

C API Languages can access the following C API below using different interface styles:

```
typedef struct {
    float a;
    float b;
} Vector;
Vector Vector_create(float a, float b) {
    return Vector { a,b };
}
Vector Vector_add(Vector a, Vector b) {
    return Vector { a.a + b.a, a.b + b.b }
}
```

Python The *ctypes* library can load shared libraries of C functions and call them from Python. Since the shared library itself does not contain type information, it requires the user to specify the type information in Python, adding syntactic overhead:

```
from ctypes import *
lib = cdll("libvector.so")
class Vector(Structure):
  _fields_ = [("a",c_float),
              ("b",c_float)]
lib.Vector_create.argtypes = [c_float,c_float]
lib.Vector_create.restype = Vector

lib.Vector_add.argtypes = [Vector,Vector]
lib.Vector_add.restype = Vector

a = lib.Vector_create(1,2)
b = lib.Vector_add(a,a)
```

LuaJIT The FFI works similarly to *ctypes*, but uses an interface where C declarations are passed in strings. However, its parser for C is not powerful enough to handle full C header files since it does not run a preprocessor.

```
ffi = require("ffi")
ffi.cdef [[
typedef struct {
    float a;
    float b;
} Vector;
Point Vector_create(float a, float b);
Point Vector_add(Vector a, Vector b);]]
V = ffi.load("libvector.so")
a = V.Vector_create(1,2)
b = V.Vector_add(a,a)
```

Terra The C code can be imported directly with no syntactic overhead. Since it loads the implementation as well, it has the opportunity to inline C functions when beneficial.

```
C = terralib.includec("vector.h")
terra main()
    var a = C.Vector_create(1,2)
    var b = C.Vector_add(a,a)
end
```

Figure 10.4: Examples of how different languages handle calling functions defined externally.

We explicitly designed Terra so that it would be compatible with C and have zero overhead. The strategy of explicitly designing a new language so that it interoperates easily with an existing one was inspired by the design of Scala, which was explicitly designed to be compatible with Java [57]. To accomplish this goal, Terra code does not require a managed runtime with a garbage collector, and is evaluated separately from the Lua runtime. Higher-level features, such as exotypes or macros, are all evaluated during typechecking, and desugared into simple data-structures and function calls. In this way, we can harness the power of existing low-level code while also providing Terra as a better interface for meta-programming.

10.2 With Lua

We have already discussed the interactions between Terra and Lua that make it possible for Lua to meta-program Terra code and types dynamically. We also use features existing in the implementation of LuaJIT to enable interaction between Lua and Terra's runtime.

Interaction across two very different languages brings up the challenges of managing the lifetime of objects across the two languages, and accessing data from one language in the other. Our approach to object management builds on Lua's model, which already has good solutions for working with low-level languages. In Lua, all objects are allocated in a garbage-collected heap. But in addition to Lua data-structures such as tables, it is also possible to allocate `userdata` on this heap, which is a special Lua type that is simply stored as an array of arbitrary bytes. This data will be collected when it is no longer reachable, and can optionally have a meta-method (`__gc`) associated with that gets called when the object is freed. This datatype allows the programmer to store Terra objects in the Lua environment. It is also possible for Lua to store pointers to external data, or "light" `userdata`. These pointers can point into a manually-managed heap such as the one used by `malloc`, but require explicitly freeing the objects. LuaJIT also extends Lua with a `cdata` type. This type holds external bytes like `userdata`, but also has an associated type in C. LuaJIT has mechanisms for introspecting into this C type to read and write data to it directly from Lua. In building off of LuaJIT, we use `cdata` objects and Terra's ABI compatibility with C to represent Terra values directly inside LuaJIT.

Lua Type	*Conversion Rule*
`number`	To any *primitive type*, using C's rules for converting a double to the primitive type.
`boolean`	To `bool`.
`nil`	To `nil`, a Terra null pointer.
`userdata`	To `&opaque`, a pointer to the value in `userdata`.
`string`	To `rawstring`, pointer to Lua's string data.
`function`	To any *function type*, a wrapper is generated that converts arguments to Terra values, invokes the function, and converts the results back to Lua.
`table`, used as array with integer keys 1 to N	To any *array type,* `T[N]`, each value of the table converted to `T`. To any *struct type,* each value in the table converted to the first `N` values in the layout of the struct. `{3,4}` *can convert to* `Complex {3, 4}`
`table`, used as a map with string keys	To any *struct type,* each value of the table converted to corresponding field in *struct type* with the same name `{imag = 3, real = 4}` *can convert to* `Complex {3, 4}`
`cdata`	To any Terra type `T` whose corresponding C type is the same as the value in the `cdata`.

Figure 10.5: Converting Lua values to Terra values.

While Lua allows references to external data to exist, it does not explicitly allow external data to reference Lua objects. Instead, the recommended way to store objects used externally is as a value in a Lua table with a known key. The external code can then use the standard Lua API functions to manipulate the table to retrieve values from the object. This design focuses on simplicity — there is no need to have a separate mechanism to hold handles to Lua objects.

Terra's model for handling data across to the two languages builds on Lua's interaction with C. Terra values can be allocated using C's malloc in a manually managed heap, or using

Lua Type	*Type Inference Rule*
`number`	If the number is representable exactly by an `int32` then it is an `int32`, otherwise it is a `double`.
`boolean`	`bool`
`nil`	`nil` type
`userdata`	`&opaque` type
`string`	`rawstring`, which is `&int8`
`function`	Used in a function call with argument Terra types `T1,…,TN`, the type inferred is `{T1,…TN} -> {}`.
`cdata`	The Terra type `T` whose corresponding C type is the same as the value in the `cdata`.
otherwise `(table, function value, etc.)`	Type cannot be inferred. The function `terralib.cast` can be used to annotate any Lua value with a Terra type if needed.

Figure 10.6: Inferring the types of Lua values. When a Lua value is used directly from Terra code through an escape, or a Terra value is created without specifying the type (e.g., `terralib.constant(3)`), then we attempt to infer the type of the object. If successful, then the standard conversion to that type is applied. The inference depends on the type of the Lua value. If none of the normal rules apply but the value has a metamethod `__toterraexpression`, this method will be called on the Lua object to generate the Terra expression before type inference occurs.

Terra Type	*Lua Type*
primitive, representable by double	`number`
primitive, larger than double	`cdata` with C type that corresponds to the Terra type
pointer, vector	Value `cdata` with C type that corresponds to the Terra type
array, struct	Reference `cdata` with C type that corresponds to the Terra type. These objects are boxed in Lua, so they will be copied in Lua by reference.

Figure 10.7: Converting Terra values to Lua values. When converting Terra values back into Lua values, the following rules apply. In cases where there is not a Lua type that can completely represent the value, it is kept as a Terra value stored in LuaJIT's `cdata` type.

the function `terralib.new(T)`, which allocates them as `cdata` managed by the Lua garbage collector.

Lua values are not directly accessible from Terra. When Terra code refers to a Lua value during specialization, or a Lua value is passed to a Terra function as an argument, it is necessary to *convert* values from Lua into Terra. Figure 10.5 describes the rules for converting Lua values to Terra. Internally, we implement this conversion on top of LuaJIT's foreign-function interface, which makes it possible to call C functions and use C values directly from Lua. Since Terra's type system is similar to that of C's, we can reuse most of this infrastructure. If the resulting Terra type is not specified (e.g., when a Lua value is used as the result of an escape), then it is inferred using the rules in Figure 10.6.

In general, primitive types are mapped to their equivalents, while Lua tables are converted into arrays or structs whose static lengths or field names match the dynamic keys and lengths of the input table. Almost all of these rules are adapted from LuaJIT's interface for calling C functions, and Terra internally uses this interface for its implementation. Note that data already allocated as Terra values are stored in LuaJIT `cdata` objects, and simply passed through unchanged. The rules for going in the opposite direction, that is converting Terra values back to Lua, are summarized in Figure 10.7.

LuaJIT also provides functionality that allows Lua code to introspect C objects. We use a wrapper around this support to extend it to Terra objects. Lua code can read and set fields of C/Terra objects directly. This functionality is useful when writing initialization code that does not need to be fast, but does need to prepare data-structures for future high-speed components. In Liszt, this functionality is used to initialize memory for fields along with other field meta-data which is normally stored in the Lua environment. It simplifies the management of fields since we can rely on the Lua garbage collector to track when objects like fields are no longer needed.

Some libraries of existing Lua code might include syntax that is not valid Lua-Terra code (e.g., if it used the keyword `terra` as a variable name). To still provide backwards compatibility in this case, we purposefully expose two sets of functionality for loading code, the function `lua_load` (`load` from inside Lua itself) which comes from Lua and parses plain Lua code, and `terra_load` (`terralib.load`), which is provided by Terra and parses Lua code with Terra declarations embedded inside of it. Once loaded this code can interact in all the normal ways with combined Lua and Terra code.

10.3 With DSLs

Many of the problems that Terra faces when interfacing with Lua and existing code are also faced by the DSLs themselves. The Lua-Terra design encourages DSLs to interact with Lua in similar ways.

The syntax extensions presented in Chapter 9 explicitly enforce a parsing phase, where only the syntax is available, and a separate runtime phase, where the lexical environment is always passed in as an argument. The easily accessible lexical environment encourages DSL authors to write languages that extensively use the lexical environment. The environment can be used to store language entities that would normally be in a custom symbol table for the DSL such as functions, or types. It ensures that the DSL entities are first-class Lua values, encouraging some degree of meta-programming for the DSL. It also makes it easy for certain language objects to be constructed with Lua functions. For instance, fields in the Liszt language are constructed using a Lua API rather than custom language extensions.

Since DSLs share the same lexical scope with Terra as well, it is easy for the DSL author to integrate Terra code directly into a DSL where possible. We have found this useful in two cases. In Darkroom, there were some instances where mostly stencil code had a small part that read an image using a data-dependent index, such as the semi-Lagrangian update in our fluid simulation. In these cases, Darkroom could call an external Terra function to perform this step.

In Liszt, scientists often have pre-constructed mathematical functions written in C. We are able to use our zero-overhead compatibility with C to import them as Terra functions, and then call the Terra functions from Liszt code using the shared lexical scope between Liszt and Terra. Without a shared scoped at compile time, a language like Liszt would need custom operators to import foreign symbols, making the interactions more verbose.

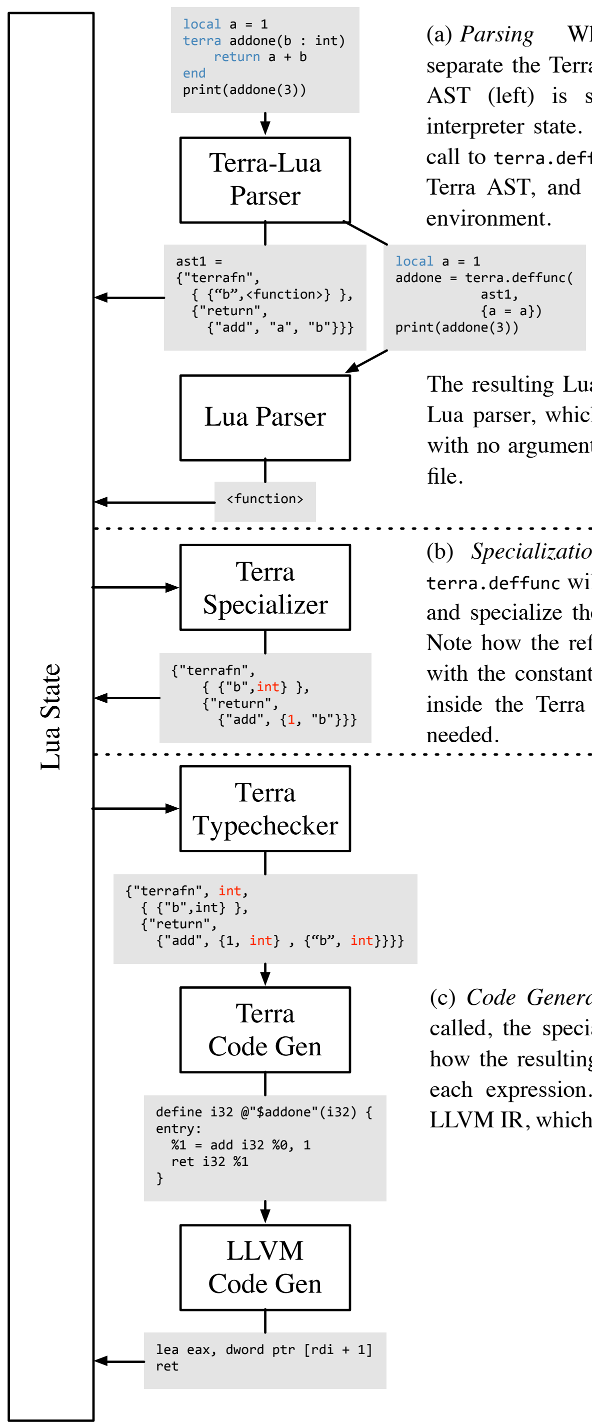
In the future, we hope that a shared compilation environment will encourage interaction between DSLs. Initially, DSLs can interoperate at the level of runtimes — one DSL being able to invoke functionality of another by calling the generated Terra functions that each produces. This level of interaction does not require the DSLs to be aware of each other to work. Later DSLs that are frequently used together can be made aware of each other and make compile-time decisions about data-structures cooperatively. The design of Terra encourages this interaction by enabling both DSLs to be compiled and run in the same process, making the meta-data needed for these choices easily available.

CHAPTER 11
IMPLEMENTATION

So far we have looked at the advantages of a two-language design for high-performance code generation from the perspective of the user of the language. We also chose a two-language design because it is substantially simpler to implement for practical reasons. There are a lot of good implementations of high-level dynamically-typed languages, and good implementations of low-level modular compilers for C. Both are widely used, and their designs have improved over time. With a two-language design, we can leverage this work without much modification. The details of executing each language well can be handled by existing libraries, allowing us to focus on details of interoperability and staging. We chose to implement our design using LuaJIT [2] for our high-level interpreter, and LLVM [49] for low-level code generation. But this same design can be easily recreated using combinations of other high- and low-level technologies.

Terra expressions are an extension of the Lua language, so our implementation augments a standard Lua runtime with additional functionality. To run plain Lua code, we use LuaJIT [2], an implementation of Lua that includes a trace-based JIT compiler. LuaJIT itself is implemented as a library in C, with calls to initialize the runtime, load Lua programs, and evaluate them. Terra adds additional functions to load combined Lua-Terra programs, and compiler infrastructure to specialize, typecheck, and generate code for Lua functions.

The way we implement the entire process is shown in Figure 11.1. Since Terra is an embedded language, the standard compiler pipeline is split into separate stages that occur when a file is (a) loaded, (b) a Terra function is specialized, and (c) a Terra function is typechecked then compiled. In the first stage, we load Lua-Terra programs. This process is implemented as a preprocessor that parses the combined Lua-Terra text. This design allows



```
local a = 1
terra addone(b : int)
    return a + b
end
print(addone(3))
```

```
ast1 =
{"terrafn",
  { {“b”,<function>} },
  {"return",
    {"add", "a", "b"}}}
```

```
local a = 1
addone = terra.deffunc(
           ast1,
           {a = a})
print(addone(3))
```

```
<function>
```

```
{"terrafn",
    { {"b",int} },
    {"return",
      {"add", {1, "b"}}}
```

```
{"terrafn", int,
  { {"b",int} },
  {"return",
    {"add", {1, int} , {“b”, int}}}}
```

```
define i32 @"$addone"(i32) {
entry:
  %1 = add i32 %0, 1
  ret i32 %1
}
```

```
lea eax, dword ptr [rdi + 1]
ret
```

(a) *Parsing* When Terra code is loaded, we parse it to separate the Terra code from plain Lua code. The Terra AST (left) is stored as a data structure in Lua's interpreter state. The Terra function is replaced with a call to `terra.deffunc`, which is passed a reference to the Terra AST, and a Lua table with the location lexical environment.

The resulting Lua code (right) is loaded using the plain Lua parser, which turns it into a *thunk*, a Lua function with no arguments that can be run to evaluate the entire file.

(b) *Specialization* When the thunk is run, `terra.deffunc` will be called to create the `addone` function and specialize the Terra AST in the local environment. Note how the reference to Lua variable “a” is replaced with the constant 1. The specialized AST is then saved inside the Terra function object until typechecking is needed.

(c) *Code Generation* When the `addone` function is called, the specialized AST will be typechecked. Note how the resulting AST is now annotated with types for each expression. This result is then transformed into LLVM IR, which is JIT compiled into assembly code.

Figure 11.1: Here we illustrate how a simple Terra program gets transformed during each stage of compilation. The Terra AST and internal API calls have been simplified slightly for clarity.

us to implement Terra without having to modify the LuaJIT implementation. The preprocessor parses the text, building an AST for each Terra function. The AST is stored in the Lua state. The preprocessor then replaces the Terra function text with a call (`terralib.deffunc`) to create the Terra function and specialize it in the local environment. This function takes as arguments a reference to the parsed AST stored in the Lua state, as well as a table that holds the local lexical environment. After preprocessing, the plain Lua code is then loaded into the Lua interpreter. Like normal Lua programs, our implementation loads Lua-Terra code into a *thunk*, a Lua function that has no arguments which is called to actually run the code.

When this thunk is executed and the code to define a Terra function is run, it invokes Terra's function specializer, and returns the first-class Lua object that represents the specialized Terra function. Terra code is compiled when a Terra function is typechecked the first time it is run. During typechecking, the compiler will evaluate any exotype properties needed by calling user-provided property functions.

We use LLVM [49] to compile Terra code since it can JIT-compile its intermediate representation directly to machine code. To implement backwards compatibility with C, we use Clang, a C frontend that is part of LLVM. Clang is used to compile the C code into LLVM and generate Terra function wrappers that will invoke the C code when called. LLVM also handles other compiler details such as emitting object files, shared libraries, and executables.

Using LLVM allows us to leverage common low-level optimization passes to get good performance. In Clang and LLVM, most optimization of C code occurs after it has been translated to LLVM. We have found that translating Terra directly to LLVM IR, and then running standard optimization passes allows us to match the speed of equivalent C programs without implementing our own set of Terra-specific optimizations.

With LLVM, we can easily target architectures other than x86 as well. Our GPU backend allows Terra code to run on CUDA-enabled devices. It uses the same process to generate LLVM IR as our CPU backend and shares the same optimization passes used to produce optimized IR. We then use LLVM's NVPTX backend to translate the LLVM IR to PTX IR, which is given to the CUDA runtime and linked against pre-existing PTX libraries for operators such as CUDA's `malloc` and `printf`. Our GPU implementation also handles the details of loading this code and generating wrapper functions for kernels that marshal the arguments and invoke the kernel.

Similar to our approach to low-level programming on the CPU, our GPU backend does not attempt to abstract over the GPU's hardware or attempt to automate the conversion of CPU code to GPU code. Instead, Terra functions used as CUDA kernels are written similarly to kernels written in CUDA C. They explicitly manage their own memory layout, and can use built-in functions to retrieve the local thread and block IDs while running. This approach allows users to match the performance of CUDA C, while gaining the benefits of meta-programming and runtime code generation provided by Terra. A simple micro-benchmark based on the SAXBY kernel from BLAS runs the same speed (within 1%) when written in Terra as equivalent code written with the native CUDA compiler (`nvcc`). One of the reasons that Terra code can perform similarly is that `nvcc` itself is internally based on LLVM and uses the same backend.

Our implementation of Terra uses around 15,000 lines of code, of which around 5,000 are written in Lua (the specializer, typechecker, and `terralib` library code), and the rest are written in C++ (parser, LLVM code generator, and interface with `clang`). For comparison, vanilla Lua 5.2—a simple interpreter design—is around 20,000 lines, and LuaJIT is 60,000 lines. Terra is comparatively smaller since it does not need to handle details such as garbage collection (handled by LuaJIT), standard library maintenance (handled through backwards compatibility with C), or low-level code generation (handled by LLVM). In contrast, a single language design for high-performance code generation could end up being significantly more difficult to create since its design would not map directly to either a high-level language, or a low-level library like LLVM. While it is hard to speculate about the complexity of such a language, other languages that are higher level than C but use LLVM as a backend such as Rust (110,000 lines), and Julia (35,000 lines) are substantially larger.

CHAPTER 12
DISCUSSION

The contributions shown in this thesis and our experience developing the Terra language suggest some areas of future work, as well as different perspectives on using Terra as a programming language for building DSLs.

12.1 LESSONS LEARNED FROM BUILDING DSLS

Darkroom, Quicksand, and Liszt were a collaboration between myself and others, who are developing these systems as independent research projects. Having collaborators build the DSLs as actual research projects ensured that they were solving realistic problems. It also gave us an opportunity to see how actual DSL writers would use Terra in practice. This collaboration resulted in improvements to the language to make particular kinds of DSL development easier.

Darkroom, developed by James Hegarty, was the first DSL we created using Terra. It stressed the ability to generate generic code and the interfaces for embedding another language inside Lua. It led us to make the interface for generating code more concise and predictable. Originally, Terra used a lazy form of specialization, which would occur at the same time as typechecking. This design made type information available at specialization time, but made it hard to know when values would be captured. The difficulty of explaining these semantics encouraged the switch to eager specialization and lazy typechecking. Darkroom also encouraged the addition of low-level memory operations, including unaligned or volatile loads and stores. This functionality is implemented as built-in macros for doing annotated loads and stores. As the first language to interoperate between Terra and Lua, it also encouraged a more seamless view of Terra's objects from Lua. We added the ability to invoke methods on Terra objects directly from Lua. Internally, this mechanism uses the exotype `__getmethod` property, and calls the result on the object directly from Lua. Finally, we added the long form quotation syntax:

```
quote
  stmts
in exp end
```

The short quote (`` `exp``), is defined via desugaring to `quote in exp end`. Unlike functional languages, Lua makes a distinction between statements and expressions. Terra inherits this syntax and behavior. The long form quote combines a quote operator with the semantics of a "let" expression from functional languages so that a programmer can include statements in a quotation. Making the quotation more general is simpler than the previous design, which required one quote operator for expressions and another for statements.

Quicksand, developed by Daniel Ritchie, was the first library to independently use the exotype interface. Exotypes allowed for expressive classes, but did not help with the automated management of object lifetimes that is possible in languages such as C++. In C++, objects allocated on the stack can have destructors that run when the stack unwinds. While class systems were implementable with exotypes, Terra lacked a mechanism for automatically scheduling a destructor to run when a scope ends. We added a `defer` statement to implement this mechanism. A statement, `defer foo(arg0,...,argN)`, schedules the function call `foo` to run when the surrounding scope exits. It is similar to the `defer` statement in Google Go, but the deferred code runs when a C++ destructor would run, rather than at the end of a function. These defer statements can be generated by macros, making it possible for library-based class system to create objects whose lifetime on the stack can be managed automatically.

Liszt, developed primarily by Crystal Lemire, Gilbert Bernstein, and Chinmayee Shah, tested our implementation of GPU primitives. It encouraged a more robust interface to GPU features such as CUDA-provided mathematical functions (e.g., `log`). It also required access to GPU-specific features including shared and constant memory. These features are expressed in the NVPTX backend as globals with special memory spaces. To use these globals generically, they need to be converted to pointers in the unified address space. Both creating the global variable, and converting it to the unified address space were possible to write in Terra, but each reference to these globals needed to be annotated with the address space conversions. To automate the process and make it invisible to the user, we added the `__toterraexpression` metamethod, which allows any Lua object to define how it should be converted into Terra code when used in an escape. Using this mechanism, the CUDA backend code creates custom Lua objects to represent shared and constant memory that automatically insert the right memory space conversions. This allows the GPU backend to support these features without modifying the rest of the compiler.

12.2 Software development as a library

When we designed applications using Lua-Terra, we found that one of the ways that the system reduces complexity is by replacing separate processes with a single process orchestrated by Lua. In Liszt and matrix-multiply, this single process replaced a lot of "glue" code written in shell scripts and Makefiles that called pre-processors or compilers. These standalone tools were originally designed to be invoked by hand, and later by build tools like `make` that simplify a repeated process. But they are not designed with the intention of being heavily meta-programmed.

Lua and Terra provide a more programmable interface to these same software development tools. The different phases of evaluation of a Terra program each roughly correspond to and replace a single program in the previous model. Lua evaluation replaces the top-level shell script or Makefile which manages the state and dependencies. Specialization of Terra code from Lua replaces the need for source-to-source translators or preprocessors. Type-checking corresponds to actually invoking the compiler to build code, and Terra evaluation corresponds to actually running the compiled code as its own program. Making these phases programmable does not prevent the use of Lua and Terra like a traditional compiler stack since it is still possible to write compiled code out to a file. Instead, its increased flexibility makes more complicated designs like DSLs or autotuners possible to write.

While Lua and Terra replace many existing tools, there are other tools that provide valuable insights into software development that still exist primarily as separate processes designed to be run by hand. Tools for debugging (e.g., `gdb`) and performance profiling (e.g., `prof` or `VTune`) provide valuable insights. They include powerful functionality. A debugger has libraries that can start and stop code at particular source locations, watch for changing expressions, and allow for runtime introspection of program state. Performance monitoring programs know how to sample hardware performance counters and quickly associate those locations with program text. This functionality can be very powerful if used programmatically. A DSL might automatically annotate kernels it generates with performance monitoring code and present an easy-to-understand view of how the DSL was performing in the terminology of that language. A DSL might also want to expose a debugger that can step through its statements, and introspect on intermediate values.

The functionality to actually do these tasks requires sophisticated architecture-specific infrastructure, and while debuggers and profilers for low-level languages include this functionality, it is not exposed in a way that is easily accessible apart from using the standalone tools by hand. Some libraries, such as PAPI [8], provide an interface to performance timing functionally, but their interface is at the level of assembly language. One barrier to a higher-level interface is the need to expose the link between performance counters and program code as part of the library. In traditional languages, this link is difficult to provide since the code only exists as strings, which were consumed in a separate process. In a system like Terra, these tools might be easier to create since it already represents code as a first-class entity in the same process that is running. This way of profiling and debugging code is familiar to the dynamically-typed language community and could provide a way to make performance profiling and debugging easier if implemented at a low-level in a system like Lua-Terra.

12.3 Statically-typed languages as heterogeneous staged programming

Terra appears different from other traditional statically-typed languages since its constructs are always created using Lua, which is dynamically typed. Nevertheless, it is possible to think of *every* static-typed language as containing multiple languages. One language, at the top level, instantiates entities like types, function definitions, classes and methods. This language “evaluates” when the compiler for that language is run producing a second “stage” containing the actual program. That program is described with a separate language which can appear in the body of functions. This view is reflected in the fact that statically-typed languages typically make a distinction between what can appear at the top-level of a file, which we might term a *declaration* language, and the expressions allowed in the body of functions, which we traditionally think of as writing code in that language.

For instance, one can think of the top-level declaration language of C as a program that defines types and global symbols, and then checks that the programs in the body of the functions have the correct types. As a programming language itself, C's declaration language is impoverished. It can introduce "variables" for types (`typedef`), and functions (function declarations). It can assign a function variable a value (function definition), and it can define simple aggregate types. The fact that this language is so limited leads to a lot of abuse of the preprocessor to add additional functionality, such as using `#if` statements to introduce conditional compilation into declarations.

Other statically-typed languages build more functionality directly into their own versions of the declaration language. C++ and Java include object-oriented features by extending the declaration language with statements to introduce classes or add methods. Most of the complexity of these object-oriented systems lies in the declaration language (the only thing added to the computation language is normally a method invocation expression).

One drawback of this design is that the object-oriented features (e.g., inheritance, traits, interfaces, etc.) are very specific and built-in to the language. Since C++ introduced classes there have arguably been improvements in the way we handle features such as multiple inheritance or interfaces such as mixins, which help solve the diamond inheritance problem [6]. But since C++'s class system is part of the language itself, it cannot easily adapt these improvements without breaking backwards compatibility. Similarly, early versions of Java lacked genetics, leading to a lot of repetitive unsafe code. Reacting to this and other limitations in Java, Guy Steele noted that any new programming language should have some plan for growth [68]. In addition to having very specific built-in features, most declaration languages are not really complete languages — typically there are no explicit loops, conditionals, or data structures, which can restrict powerful meta-programming behaviors that using a full language allows.

Rather than continue to add specific features to this declaration language and deal with its shortcomings, our approach in Terra is to replace it entirely with Lua. Doing so removes the need for many language-specific features but provides a powerful mechanism to grow the language. Namespaces are handled by storing values in Lua tables. Templates are handled by nested Terra definitions in a Lua function. Terra's exotypes are also designed with growth in mind. Exotypes provide the bare mechanisms for object-orientation (method invocation syntax) and allow the user to implement any desired functionality as a library via meta-programming.

Using a full language makes the way that these features are handled much more flexible. Templates in C++ and parameterized types in other languages can emit cryptic error messages. One reason this occurs is because the error is described in terms of the type system rather than by the developer of the template. Since templating in Terra is accomplished by a Lua function, it is possible for the developer of the template function themselves to write better and simpler to understand error messages that are emitted when it is used incorrectly.

Recognizing that statically-typed languages really are a composition of multiple languages helps demonstrate some of the limitations of using them, but there are some downsides to Terra's approach of using a full language as the declaration language. Tools that want to provide introspection on code need to actually execute a full Lua program to understand what types and methods exist. Thus, it is harder to understand code just by reading it. Having a full program also limits possibilities for performing static analysis at this level. While at first this seems problematic, it may be possible to adapt tools to this model where part of the program actually runs during development. Developers might print out details about relevant types or disassemble functions on demand to debug performance issues. During development the programmer might have the running program communicate directly with an IDE to inform it about what types and functions the declaration program is producing. Some of these interactions have already been implemented using F# type providers [71] and can be incorporated into Terra as well.

12.4 A MIDDLE GROUND FOR STATIC AND DYNAMIC TYPING

Terra exotypes are neither dynamically or statically typed. Unlike static languages, type-checking can occur during program execution. But unlike dynamically-typed languages, this checking occurs function-at-time when a function is first needed rather than as a function evaluates, which provides an opportunity for type errors to be caught before code is run. Our API also allows the programmer to explicitly ask for typechecking and compilation of functions using the `compile` method.

This approach to typechecking is more flexible than the traditional approach. If desired, traditional typechecking behavior can be achieved by generating all Terra code using quotations and exotypes, and then calling `compile` to typecheck the result. The code could then be saved to an object file and used later. Making typechecking of a type system an *operation* in the language itself allows us to explore a middle ground between static and dynamic typing.

Languages with gradual type systems may benefit from the mixture of static and dynamic typing. Gradual typing [64] provides a mechanism for annotating and statically checking some types in an otherwise dynamically-typed language. Values without annotations are checked when used as inputs to typed code, but internal to typed code these checks could be omitted, improving performance while maintaining safety. Any gradual typing system needs to provide some system for annotating types. A simple system would only be able to provide types for a small amount of code in a program. A more complicated system could type more of the program, but would add complexity to the programming language.

An advantage of dynamically-typed languages is their relative simplicity, and adding a complex type system can detract from their appeal. A middle ground where functions are typechecked during evaluation could keep the type system simple while still allowing for typechecking of functions before they are run. The type system could use a simple model of types, but like Terra exotypes, allow the runtime to meta-program their behavior during typechecking. Functions would be typechecked either when they are defined or when explicitly requested. Behavior similar to normal gradual typing could be achieved by factoring the program into a "top-level" that defined types and functions, and a 'main' function that would be invoked to run the program. Evaluating the top-level of the program would then define and check all the gradual types. This mixture of static and dynamic typing in a single language might allow for improved performance, better safety, or reduced complexity compared to existing approaches.

12.5 Applying dynamic code generation to scientific computing

The ability to do just-in-time compilation of code using Terra can be applied to areas that have traditionally avoided the approach due to its added complexity, such as scientific computing. In Liszt, for example, we could perform JIT compilation based on dynamic stencil analyses. The version of Liszt written in Terra can dynamically load code, but we have not fundamentally changed its approach for doing data-dependency analysis for running on large clusters.

The new design of Liszt, based on dynamic compilation, makes it possible to replace the current static analysis, which runs once per function on a static mesh, with an analysis that runs somewhat dynamically. Rather than make the stencil analysis conservative, the compiler can make aggressive assumptions, and insert guards that dynamically check these assumptions during execution. If the stencil trips one of these guards or the mesh topology changes, the kernels could be regenerated and the data structures rebuilt to adapt to the change. It is likely that this JIT compilation approach of making aggressive assumptions in combination with guards may prove useful to other fields as well. Terra's ability to easily provide runtime code generation makes it possible to explore these designs more easily.

12.6 Using high-performance staged programming to improve compiler performance

Currently compilers are designed to be either run online (e.g., javascript JITs) or as offline passes (e.g., `gcc` or `clang`). The online compilers are designed to emit code fast, and to do so they are often specialized to emit only the kind of code necessary for the language that they are emitting. Offline compilers like LLVM [49] are written in a way that is more modular, with an abstract notion of a compiler pass and a pipeline of such passes as a way to represent compilation. This modularity comes at the cost of compilation speed. In tests when developing the Riposte VM [74], we found an order of magnitude difference between compilation speed using a hand-written JIT compiler, and passing the same code to LLVM when emitting code for the common Black-Scholes benchmark. Part of this difference is due to LLVM performing more sophisticated transformations, but even when those optimizations are disabled, performance suffers due to the overhead of the modular abstractions and the data layout of the intermediate representations.

Terra's approach to staging high-performance code may make it possible to create very fast compilers that are still modular. Using exotypes, we were able to create an x86 assembler object that ran an order-of-magnitude faster than hand-written code by applying automated optimizations such as template fusion. Other parts of traditional compilers are possibly amenable to the same improvements if we can represent internal compiler transformations using a higher-level of abstraction. For instance, instruction selection can be driven by high-level templates that are translated into lower-level search procedures. LLVM already represents some instruction selection this way using its `tablegen` language. IR transformation passes can also be described at a higher level, allowing fusion of code across compiler passes to reduce the cost of traversing the IR. The data structures representing the IR itself can be optimized based on what transformations will actually be applied to it. A successful design can make the approach of JIT compiling code applicable to a much wider body of problems by improving the speed of online compilation.

CHAPTER 13
CONCLUSION

This thesis set out to make portable high-performance programming easier by creating a principled way to programmatically generate high-performance code at a low level. Our solution proposes a novel two-language design that uses the high-level language Lua to meta-program a new low-level language Terra. The two-language design reflects the way actual DSLs and autotuners are built in practice, where high-level languages are frequently used to write the compiler, while low-level languages are used at runtime.

We show how to adapt principled techniques from multi-stage programming so they would work in the context of a two-language design. Shared lexical scoping between Terra and Lua makes it easy to manage Terra entities inside of Lua, and provides a basis for their interaction. Variable hygiene makes it possible for the programmer to understand the relationship between a variable's declaration and use, even if they span two languages. We also distinguish between staging and evaluation using different semantic phases of execution. By ensuring the evaluation phase of Terra occurs independently from Lua, we ensure that programmers can control the performance of generated Terra code.

We extend this two-language design to also support the generation of flexible high-performance types. Here, we combine the advantages of each language to get the flexibility of dynamic languages to create types based on runtime information with the performance of types from statically-typed languages. We show how to combine meta-object protocols commonly used in dynamically-typed languages with staged programming techniques that run during Terra's typechecking phase.

To make the interaction of Lua and Terra clear, we also provide formal semantics for the phases of evaluation. We provide a language extension mechanism so other DSLs can be embedded in Lua in the same way we embedded Terra. And we describe the design choices we made to ensure that Lua and Terra can interoperate well with each other and with existing applications.

We evaluate our two-language design by creating example DSLs and libraries in a wide variety of areas including linear algebra, image processing, physical simulation, probabilistic computing, class system design, serialization, dynamic assembly, and automatic differentiation. We show that this software can be easily created using our design, often using fewer languages and technologies than similar software built with current tools. We also show that performance of these examples matches or exceeds the existing tools.

A common theme across our examples is that the added simplicity and expressiveness found when writing in Lua and Terra makes it feasible to implement aggressive optimizations that were not attempted in existing approaches. In the future we hope that this approach of creating concise but highly specialized libraries and languages can be used to optimize more domains, making it possible to experiment with more radical designs for software and hardware.

BIBLIOGRAPHY

[1] Google V8 Javascript engine. `http://code.google.com/p/v8/`.

[2] The LuaJIT project. `http://luajit.org/`.

[3] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: A step towards reconciling dynamically and statically typed OO languages. In DLS, pages 53–64, 2007.

[4] A. Bawden and J. Rees. Syntactic closures. In LFP, pages 86–95, 1988.

[5] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science and Engineering*, 13.2:31–39, 2011.

[6] G. Bracha and W. Cook. Mixin-based inheritance. In OOPSLA/ECOOP, pages 303–311, 1990.

[7] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In PACT, pages 89–100, 2011.

[8] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In SC, 2000.

[9] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In SCALA, pages 3:1–3:10, 2013.

[10] J. Carette. Gaussian elimination: A case study in efficient genericity with MetaOCaml. *Sci. Comput. Program.*, 62(1):3–24, Sept. 2006.

[11] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In PPoPP, pages 47–56, 2011.

[12] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In OOPSLA, pages 835–847, 2010.

[13] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In PPoPP, pages 35–46, 2011.

[14] S. Chiba. A metaobject protocol for C++. In OOPSLA, pages 285–299, 1995.

[15] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In PLDI, pages 122–133, 2010.

[16] A. Cohen, S. Donadio, M. jesus Garzaran, C. Herrmann, and D. Padua. In search of a program generator to implement generic transformations for high-performance computing. In *1st MetaOCaml Workshop (associated with GPCE)*, pages 166771–7, 2004.

[17] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In POPL, pages 493–501, 1993.

[18] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL, 1996.

[19] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, New York, 2002. Springer.

[20] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 25–39, London, 2000. Springer-Verlag.

[21] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In POPL, pages 297–302, 1984.

[22] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In SC, pages 9:1–9:12, 2011.

[23] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. In PLDI, pages 105–116, 2013.

[24] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class runtime generation of high-performance types using exotypes. In PLDI, pages 77–88, 2014.

[25] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. Swadi, and W. Taha. Implicitly heterogeneous multi-stage programming. *New Gen. Comput.*, 25(3):305–336, Jan. 2007.

[26] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In OOPSLA, pages 391–406, 2011.

[27] M. Flatt. Composable and compilable macros: you want it when? In ICFP, pages 72–83, 2002.

[28] T. Foley and P. Hanrahan. Spark: Modular, composable shaders for graphics hardware. In SIGGRAPH, pages 107:1–107:12, 2011.

[29] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In PLDI, pages 315–326, 2005.

[30] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216 –231, 2005.

[31] Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, Dec. 1999.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, 1995.

[33] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008.

[34] Google. Protocol buffers.

[35] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theor. Comput. Sci.*, 248(1-2):147–199, Oct. 2000.

[36] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectJ. In OOPSLA, pages 161–173, 2002.

[37] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. In SIGGRAPH, pages 144:1–144:11, 2014.

[38] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In PLDI, pages 326–336, 1994.

[39] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua — an extensible extension language. *Software: Practice and Experience*, 26(6), 1996.

[40] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of Lua. In HOPL III, pages 2:1–2:26, 2007.

[41] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes. Passing a language through the eye of a needle. *Commun. ACM*, 54(7):38–43, July 2011.

[42] O. Inc. Java object serialization specification. `http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html`.

[43] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. *SIGPLAN Not.*, 20(8):82–87, Aug. 1985.

[44] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[45] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[46] O. Kiselyov and C. Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384, 2009.

[47] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP, pages 151–161, 1986.

[48] J. Lamping, G. Kiczales, L. H. Rodriguez, Jr., and E. Ruf. An architecture for an open compiler. In *Proc. of the IMSA'92 Workshop on Reflection and Meta-Level Architectures*, pages 95–106, 1992.

[49] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In CGO, 2004.

[50] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[51] G. Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In ICFP, pages 311–322, 2012.

[52] J. McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, Aug. 1978.

[53] R. Meyers. X macros. *C/C++ Users J.*, 19(5):52–56, May 2001.

[54] R. Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, pages 113–162, 2011.

[55] G. Neverov and P. Roe. Metaphor: A multi-staged, object-oriented programming language. In GPCE, 2004.

[56] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In CGO, pages 224–235, April 2011.

[57] M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, and et al. An overview of the Scala programming language. Technical report, 2004.

[58] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM Trans. Program. Lang. Syst.*, 21(2): 324–369, Mar. 1999.

[59] V. R. Pratt. Top down operator precedence. In POPL, pages 41–51, 1973.

[60] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):21–45, Feb. 2004.

[61] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. In SIGGRAPH, 2012.

[62] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In GPCE, pages 127–136, 2010.

[63] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In POPL, pages 497–510, 2013.

[64] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[65] J. M. Siskind. Flow-directed lightweight closure conversion. Technical report, 1999.

[66] J. Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, March 2003.

[67] Stan Development Team. Stan: A C++ library for probability and sampling, version 1.3, 2013. URL `http://mc-stan.org/`.

[68] G. L. Steele, Jr. Growing a language. In OOPSLA Addendum, pages 0.01–A1, 1998.

[69] B. Stroustrup. Multiple inheritance for C++. In *Proceedings of the of the Spring 87 European Unix Systems Userss Group Conference*, 1987.

[70] N. Sweet. Kryo. `https://code.google.com/p/kryo/`.

[71] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. F#3.0 — strongly-typed language support for internet-scale information sources. Technical report, 2012. URL `http://research.microsoft.com/apps/pubs/?id=173076`.

[72] W. Taha. A Gentle Introduction to Multi-stage Programming. In *Domain-Specific Program Generation*, pages 30–50. 2004.

[73] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, pages 203–217. ACM Press, 1999.

[74] J. Talbot, Z. DeVito, and P. Hanrahan. Riposte: a trace-driven compiler and parallel vm for vector code in r. In PACT, pages 43–52, 2012.

[75] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In PLDI, pages 132–141, 2011.

[76] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In ICFP, pages 146–156, 2001.

[77] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Softw. Pract. Exper.*, 35(2):101–121, Feb. 2005.

[78] D. Wingate, A. Stuhlmüller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proc. of the 14th Artificial Intelligence and Statistics*, 2011.