Terra: A Multi-Stage Language for High-Performance Computing

Zachary DeVito James Hegarty Alex Aiken Pat Hanrahan Jan Vitek
Stanford University Purdue University
(zdevito|jhegarty|aiken|hanrahan)@cs.stanford.edu jv@cs.purdue.edu

Abstract

High-performance computing applications, such as auto-tuners and domain-specific languages, rely on generative programming techniques to achieve high performance and portability. However, these systems are often implemented in multiple disparate languages and perform code generation in a separate process from program execution, making certain optimizations difficult to engineer. We leverage a popular scripting language, Lua, to stage the execution of a novel low-level language, Terra. Users can implement optimizations in the high-level language, and use built-in constructs to generate and execute high-performance Terra code. To simplify metaprogramming, Lua and Terra share the same lexical environment, but, to ensure performance, Terra code can execute independently of Lua's runtime. We evaluate our design by reimplementing existing multi-language systems entirely in Terra. Our Terra-based auto-tuner for BLAS routines performs within 20% of ATLAS, and our DSL for stencil computations runs 2.3x faster than hand-written C.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code Generation, Compilers

General Terms Design, Performance

Keywords Lua, Staged computation, DSL

1. Introduction

There is an increasing demand for high-performance power-efficient applications on devices ranging from phones to supercomputers. Programming these applications is challenging. For optimum performance, applications need to be tailored to the features of the target architecture, e.g., multi-core, vector instructions, and throughput-oriented processors such as GPUs. Applications have turned to generative programming to adapt to complex hardware. Auto-tuners like SPIRAL [23], ATLAS [33], or FFTW [12] can express a range of implementations for specific applications such as FFTs, and choose the best optimizations for a given architecture. In areas such as machine learning [4], or physical simulation [9], domain-specific languages (DSLs) can achieve the same goal for a range of similar applications through domain-specific optimizations.

However, high-performance applications that rely on code generation are often implemented as ad hoc source-to-source translators. For instance, consider FFTW which implements its `genfft` compiler in OCaml and emits C code [12], or Liszt, a DSL which uses Scala for its transformations and generates code that links against a runtime written in C [9].

While these designs produce high-performance code, they are hard to engineer. A DSL or auto-tuner typically has three components: an optimizer that performs domain-specific transformations to generate a plan of execution, a compiler that generates high-performance code based on the plan, and a runtime that supports the generated code and provides feedback to the optimizer. If, as in FFTW and Liszt, the optimizer and compiler are separate from the runtime, it is difficult to feed runtime statistics back to the compiler to perform problem-specific optimizations. Transformations also require careful engineering to separate compile-time and runtime optimizations, making it difficult to prototype new optimizations.

Ideally, it should be easy for domain experts to experiment with domain and problem-specific transformations, generate high-performance code dynamically, and provide runtime feedback to improve performance. Furthermore, all parts of the toolchain, compiler, generated code, and runtimes, should inter-operate amongst themselves and with legacy high-performance libraries. Achieving these goals in a single system is complicated by the fact that each component has different design constraints. It is easier to prototype compiler transformations in an expressive high-level language, but achieving high performance in the generated code and runtime requires fine-grained control over execution and memory resources, which is easier in a low-level language.

To address these problems, we use multi-stage programming [30] to couple an existing high-level language, Lua, with a new low-level language, Terra. Lua is a high-level dynamically-typed language with automatic memory management and first-class functions [14]. Terra, on the other hand, is a statically-typed language similar to C with manual memory management. Terra code is embedded in Lua. Using multi-stage programming, programmers can generate and execute Terra code dynamically from Lua.

This two-language design allows domain experts to experiment with high-level transformations using Lua, while still generating high-performance code using Terra. To simplify code generation, the evaluation of Lua and the *generation* of Terra code share the same lexical environment and variable references are hygienic across the two languages. To ensure fine-grained control of execution, Terra executes in a *separate* environment: Terra code runs independently of the Lua runtime. It can run in a different thread, or (in the future) on accelerators like GPUs. This separation ensures that the high-level features of Lua do not creep into the execution of Terra. Furthermore, Terra exposes the low-level features of modern hardware such as vector instructions. Finally, we leverage the fact that Lua was specifically designed to be embedded in low-level languages such as C [15]. Lua's stack-based C API makes it easy to interface with legacy code, while a built-in foreign-function interface [1] makes it possible to pass values between Lua and Terra.

Furthermore, we provide support for type reflection on Terra types that enables the creation of new types via meta-programming. This design keeps the Terra language simple while still allowing the creation of libraries to implement higher-level components such as class systems that can be used in high-performance runtimes.

This paper makes the following contributions:

- We present the design of Terra which uniquely combines the staging of a low-level language using a high-level one, shared lexical scoping, separate evaluation, and type reflection.[1]
- We provide a formal semantics of core Terra to elucidate the interaction between Terra and Lua, focusing on how staging operates in the presence of side effects in Lua.
- We show that we can reimplement a number of existing multi-language systems entirely in Terra, but still achieve similar performance. In particular, we show an auto-tuner for matrix multiply that performs within 20% of ATLAS, but uses fewer than 200 lines of Terra code, and we present a stencil computation language that performs 2.3x faster than hand-written C. Finally, we implement a class system and container with parameterizable data layout as JIT-compilable Terra libraries, which would be difficult to engineer in existing languages.

2. Writing Multi-stage Code in Terra

We use an example image-processing algorithm to introduce Terra. At the top-level, a program executes as Lua, augmented with constructs to create Terra functions, types, variables, and expressions. The `terra` keyword introduces a new Terra function (Lua functions are introduced with `function`):

```
terra min(a: int, b: int) : int
  if a < b then return a
  else return b end
end
```

Terra functions are lexically-scoped and statically-typed, with parameters and return types explicitly annotated. In contrast, Lua has no type annotations. Terra is also backwards-compatible with C:

```
std = terralib.includec("stdlib.h")
```

The Lua function `includec` imports the C functions from `stdlib.h`. 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"]`. So, for example, `std.malloc` is C's `malloc`.

Terra entities (functions, types, variables and expressions) are first-class Lua values. For example, the follow statement constructs a Terra type that holds 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 `floats`, 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).

[1] Our implementation and additional examples are available at `github.com/zdevito/terra`

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 (e.g., `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 the `init` function, we call `std.malloc` to allocate memory for our image. Since `std` is a Lua table, Terra will evaluate the table select operator (`std.malloc`) during compilation and resolve it to the `malloc` function. 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.

Alternatively, we can save the Terra function to a `.o` file which can be linked to a normal C executable:

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

We may want to optimize the `laplace` function 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 brackets (`[]`) around the expression are the Terra equivalent of the *escape* operator from multi-stage programming, allowing a value evaluated in Lua (the code for the loop nest generated by `blockedloop`) to be spliced into the Terra expression. The third argument to `blockedloop` 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 creates a *quotation*, a block of Terra code that can be spliced into another Terra expression. Here, we use it to create the loop body using the loop indices.

The implementation of `blockedloop` walks through the list of `blocksizes`. 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:

```
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
```

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. 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. 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. We will use this formalism in Section 4.1 to illustrate key design decisions in Terra.

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 ($\xrightarrow{S}$), a process analogous to macro expansion in LISP that evaluates 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:

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

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 ($\mathtt{tdecl}$) 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:

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

Alternatively, $\mathtt{tdecl}$ creates just a declaration that can be defined later:

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

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:

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

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

$$\underline{\dot{\mathsf{e}}} \quad ::= \quad \mathsf{b} \mid \underline{\dot{\mathsf{x}}} \mid \underline{\dot{\mathsf{e}}}(\underline{\dot{\mathsf{e}}}) \mid \mathsf{tlet}\ \underline{\dot{\mathsf{x}}} : \dot{\mathsf{T}} = \underline{\dot{\mathsf{e}}}\ \mathsf{in}\ \underline{\dot{\mathsf{e}}} \mid l$$

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 $\underline{\dot{\mathsf{x}}}$.

The judgment $\mathsf{e}\ \Sigma_1 \xrightarrow{L} \mathsf{v}\ \Sigma_2$ 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)

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

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

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

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

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

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

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

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

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

Figure 1. The rules $\xrightarrow{L}$ for evaluating Lua expressions.

to Terra functions. Terra functions can be defined ($\langle \underline{\dot{\mathsf{x}}}, \dot{\mathsf{T}}, \dot{\mathsf{T}}, \underline{\dot{\mathsf{e}}}\rangle$), or undefined ($\bullet$). Figure 1 defines the Lua evaluation rules. We use two notational shortcuts:

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

Rule LTDECL creates a new Terra function at address l and initializes it as undefined ($\bullet$). Rule LTDEFN takes an undefined Terra function (e_1) and initializes it. First, e_2 and e_3 are evaluated as Lua expressions to produce the type of the function, $\dot{\mathsf{T}}_1 \to \dot{\mathsf{T}}_2$. The body, $\dot{\mathsf{e}}$, is specialized. During specialization, Terra variables (x) are renamed to new symbols ($\underline{\dot{\mathsf{x}}}$) to ensure hygiene. Renaming has been previously applied in staged-programming [30] and hygienic macro expansion [2]. In the case of LTDEFN, we generate a fresh name $\underline{\dot{\mathsf{x}}}$ for the formal parameter x, and place it in the environment. Variable x will be bound to the value $\underline{\dot{\mathsf{x}}}$ in the scope of any Lua code evaluated during specialization of the function. During specialization, Rule SVAR will replace uses of x in Terra code with the value of x in the environment.

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

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

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

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

Figure 2. The rules $\xrightarrow{S}$ for specializing Terra expressions.

$$\mathsf{b}\ \dot{\Gamma}, F\ \xrightarrow{T}\ \mathsf{b} \quad \text{(TBAS)} \qquad l\ \dot{\Gamma}, F\ \xrightarrow{T}\ l \quad \text{(TFUN)}$$

$$\underline{\dot{\mathsf{x}}}\ \dot{\Gamma}, F\ \xrightarrow{T}\ \dot{\Gamma}(\underline{\dot{\mathsf{x}}}) \quad \text{(TVAR)}$$

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

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

Figure 3. The rules $\xrightarrow{T}$ for evaluating Terra expressions.

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

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

Figure 4. Typing rules for references to Terra functions.

Rule LTAPP describes how to call a Terra function from Lua. The actual parameter 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 Core only allows values b of base types to be passed and returned from Terra functions (full Terra is less restricted).

Figure 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 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. They ensure all functions that are in the connected component of a function are typechecked before the function is run.

4. Key Design Decisions

We want to make it easier to prototype domain- and problem-specific transformations, dynamically compile the results of the transformations into high-performance code, and support this code with high-performance runtime libraries. We highlight some important design decisions in the semantics of Terra Core that make these goals possible. We then present engineering decisions that also address these issues.

4.1 Language Design

Hygienic staged programming with a shared lexical environment. The combination of staged programming, shared lexical environment, and hygiene provides several benefits. The staged programming of Terra from Lua provides interoperability between compiler, generated code, and runtime of a DSL. DSL compilers written in Lua can generate arbitrary code using a combination of quotations, escapes, and `terra` definitions. The shared lexical environment makes it possible to organize Terra functions in the Lua environment, and refer to them directly from Terra code without explicit escape expressions. To further reduce the need for escape expressions, we also treat lookups into nested Lua tables of the form $\mathtt{x.id_1.id_2...id}_n$ (where $\mathtt{id_1...id}_n$ 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.malloc`), removing the need for an explicit namespace mechanism in Terra. Finally, 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.

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 $\mathtt{x}$ to Lua values $\mathtt{v}$. Terra *syntax* $\dot{\underline{\mathtt{e}}}$ is one type of Lua *value*. This example illustrates the shared environment:

$$\begin{array}{l}\mathtt{let\ x_1 = 0\ in}\\ \mathtt{let\ x_2 =\grave{}\ (tlet\ y_1 : int = 1\ in\ x_1)\ in}\\ \mathtt{let\ x_3 = ter\ \ tdecl(y_2 : int) : int\ \{\ x_2\ \}\ in\ x_3}\end{array}$$

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

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

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

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

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

$$\begin{array}{l}\mathtt{let\ x_1 = fun(x_2)\{\grave{}tlet\ y : int = 0\ in\ [x_2]\}\ in}\\ \mathtt{let\ x_3 = ter\ \ tdecl(y : int) : int\ \{\ [x_1(y)]\ \}\ in\ x_3}\end{array}$$

The variable $\mathtt{y}$ on line 2 is introduced by the Terra function definition. It is referenced by the Lua expression inside the escape ($[\mathtt{x_1(y)}]$). The variable $\mathtt{y}$ is then passed as an argument to Lua function $\mathtt{x_1}$, 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, $\mathtt{x_3}$ would specialize to the following, causing the `tlet` expression to unintentionally capture $\mathtt{y}$:

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

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 $\dot{\underline{\mathtt{y}}}_1$ for the argument $\mathtt{y}$ binding it into the shared environment ($\Sigma[\mathtt{y} \leftarrow \dot{\underline{\mathtt{y}}}_1]$), and SLET will similarly generate the fresh name $\dot{\underline{\mathtt{y}}}_2$ for the `tlet` expression. Since $\mathtt{y}$ on line 2 has the *value* $\dot{\underline{\mathtt{y}}}_1$ during specialization, the variable $\mathtt{x_2}$ will get the value $\dot{\underline{\mathtt{y}}}_1$, and $\mathtt{x_3}$ will specialize to the following, avoiding the unintentional capture:

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

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. 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 $\mathtt{v}$ that is not also a Terra term $\dot{\underline{\mathtt{e}}}$, or resolve a Lua expression used as a Terra type to a value $\mathtt{v}$ that is not a Terra type $\dot{\mathtt{T}}$. 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 (and reflected in Terra Core), 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 compiled and when it is used. For instance, consider the following example (we use the syntax $\mathtt{e; e}$ as sugar for $\mathtt{let\ \_ = e\ in\ e}$):

$$\begin{array}{l}\mathtt{let\ x_1 = 0\ in}\\ \mathtt{let\ y = ter\ \ tdecl(x_2 : int) : int\ \{\ x_1\ \}\ in}\\ \mathtt{x_1 := 1;}\\ \mathtt{y(0)}\end{array}$$

Since specialization is performed eagerly, the statement $\mathtt{y(0)}$ will evaluate to 0. In contrast, if specialization were performed *once* lazily, then it would capture the value of $\mathtt{x_1}$ the first time $\mathtt{y}$ is called and keep that value for the rest of the program, which would lead to surprising results (e.g., if $\mathtt{y}$ were used before $\mathtt{x_1 := 1}$ 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 defined before it is used, 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:

$$\begin{array}{l}\mathtt{let\ x_2 = tdecl\ in}\\ \mathtt{let\ x_1 = ter\ \ tdecl(y : int) : int\ \{\ x_2(y)\ \}\ in}\\ \mathtt{ter\ \ x_2(y : int) : int\ \{\ x_1(y)\ \};}\\ \mathtt{x_1(0)}\end{array}$$

Alternatively, we could have provided a form of Terra definition that allows the definition of multiple mutually-recursive functions at one time. However, this approach does not inter-operate well with generative programs such as a DSL compiler that may need to create an arbitrarily sized connected-component based on dynamic information.

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 $\mathtt{x}_1$ when it is defined if $\mathtt{x}_2$ was given a type during declaration. However, even though $\mathtt{x}_1$ would typecheck, we would still receive a *linking* error if $\mathtt{x}_1(0)$ occurred before the definition of $\mathtt{x}_2$. So performing typechecking eagerly would not reduce the number of places an error might occur for function $\mathtt{x}_1$. 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.

In the full Terra language, performing typechecking lazily also provides several advantages. Forward declarations of functions do not have to have type annotations making them easier to maintain, and user-defined `struct` types do not need all their members or methods specified before being used in a Terra function. In the default case, we can keep type-checking monotonic by ensuring that members and methods can only be added to user-defined types and not removed. In actuality, the mechanisms for type-reflection described later in this section allow user-defined libraries to override the default behavior of a type (e.g., by adding inheritance). In this case, to maintain monotonic typechecking, implementors must ensure that the functionality of a type only grows over the execution of the program.

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:

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

The Terra function will specialize to $\langle \underline{\dot{\mathtt{x}}}, \mathtt{int}, \mathtt{int}, 1\rangle$, so the function call will evaluate to the value 1, despite $\mathtt{x}_1$ being re-assigned to 2. An alternative design would allow Terra evaluation to directly refer to $\mathtt{x}_1$. For instance, in MetaOCaml [29], `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. 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 port Terra to new architectures such as GPUs, run code in multiple threads, or link code into existing C programs without including the Lua runtime.

Mechanisms for type reflection. Terra is a low-level monomorphic language. Its simplicity makes it easier to implement, but can make programming libraries such as DSL runtimes tedious. For instance, a DSL writer may want to experiment with different data layouts such as array-of-structs or struct-of-arrays. Instead of adding this functionality to Terra, we provide a type reflection API for creating and examining Terra types so this higher-level functionality can be implemented as libraries. The basis of the API lies in the fact that Terra types are Lua values, as illustrated in Terra Core:

$$\begin{array}{l}\mathtt{let}\ \mathtt{x}_3 = \mathtt{fun}(\mathtt{x}_1)\{\mathtt{ter\ tdecl}(\mathtt{x}_2 : \mathtt{x}_1) : \mathtt{x}_1\ \{\ \mathtt{x}_2\ \}\}\ \mathtt{in} \\ \mathtt{x}_3(\mathtt{int})(1)\end{array}$$

The Lua function $\mathtt{x}_3$ will generate a Terra identity function for any given type. Here we call it with `int`, which will result in the specialized Terra function $\langle \underline{\dot{\mathtt{x}}}, \mathtt{int}, \mathtt{int}, \underline{\dot{\mathtt{x}}}\rangle$.

In the full language we supplement this behavior with an API to introspect and create types in Lua. Terra types include methods for introspection (e.g., `t:ispointer()`, or `t:isstruct()`) that can be called from Lua. Furthermore, structs can be created programmatically. In addition to the `methods` table presented in Section 2, structs also contain an `entries` table that describes their in-memory layout. Here we layout complex number type using its `entries` table directly:

```
struct Complex {}
Complex.entries:insert { field = "real", type = float }
Complex.entries:insert { field = "imag", type = float }
```

A struct also contains a `metamethods` table that can override certain compile-time behaviors. For instance, we might want to allow the promotion of a `float` to a complex number. A user-defined implicit conversion can be created using a Lua function `__cast` in the struct's `metamethod` table. During typechecking, when Terra needs to convert one type to another and no default conversions apply, it will call the `__cast` metamethod of either type to see if it could implement the conversion (if both are successful, we favor the metamethod of the starting type). The following example defines a conversion from a `float` to a complex number, `Complex`:

```
Complex.metamethods.__cast = function(fromtype,totype,exp)
  if fromtype == float then
    --valid, construct a complex number from the float exp
    return `Complex { exp, 0.f }
  end
  error("invalid conversion")
end
```

If there is a valid conversion, the method returns a quotation that implements the conversion (the back-tick is a shorthand for creating single-expression quotations). Using a Lua function to determine the behavior of conversions provides expressibility without the need for a more complicated mechanism.

To organize functions related to a particular type, we also provide a method invocation syntax `obj:my_method(arg)` that is desugared during typechecking to `[T.methods.my_method](obj,arg)`, where `T` is the static type of `obj`. The combination of these features allows many components such as polymorphic class systems to be implemented as libraries (shown later in section 6.3).

4.2 Engineering Design

A high-level language for prototyping. Lua provides automatic memory management, first-class functions, and built-in tables that make it easy to manage structures like ASTs and graphs, which are frequently used in DSL transformations. Its dynamic typing makes it easier to prototype different data structures and to construct arbitrary functions of Terra types and expressions.

A low-level language for performance. High-level programming languages can make it difficult to control when and what optimizations will take place. Auto-tuners and DSLs already capture the knowledge of how to generate high-performance code, so it is important to give them as much control as reasonable to express optimizations. We designed Terra to be a thin abstraction layer on modern processors. Terra provides much the same functionality as C including manual memory management, pointer arithmetic, and monomorphic functions. Global state, though not present in Terra Core, is possible in the full language using global variables created with the `global` function. Additionally, Terra includes fixed-length vectors of basic types (e.g., `vector(float,4)`) to reflect the presence of SIMD units on modern processors. Since the design of Terra is close to the hardware, users can more precisely express the execution behavior that they desire, and get predictable performance.

Cross-language interoperability using a foreign-function interface. In Terra Core, Lua can only pass base values to Terra functions and receive them as results. In the full language, when a Lua environment *is* available, we use LuaJIT's foreign function interface(FFI) [1] to translate values between Lua and Terra both along function call boundaries and during specialization. The similarity of Terra's type system to C's enables us to adapt the FFI to work with Terra. In addition to base types, it supports conversion of higher-level objects. For instance, Lua tables can be converted into structs when they contain the required fields. Lua functions can also be converted into Terra functions by generating wrapper code to dynamically convert the types on entry and exit. Since conversions are defined for Lua functions, calling a Lua function from Terra code is just a special case of converting a Lua value into a Terra value during specialization.

Backwards compatible with C. We believe that the lack of interoperability with existing code is a key factor limiting the adoption of DSLs. Terra can call C functions, making it possible to use existing high-performance libraries in the implementation of runtimes, and produce code that is binary compatible with C programs. Since Lua is easily embedded in C programs, it is easy to incorporate a mixed Lua-Terra program into existing C code. Since most languages have interfaces for calling C functions, this design makes it possible to use Terra in existing systems.

5. Implementation

Terra expressions are an extension of the Lua language. We use LuaJIT [1], an implementation of Lua that includes a trace-based JIT compiler. Lua itself is implemented as a library in C, with calls to initialize the runtime, load Lua programs, and evaluate them. We add additional functions to this library to load combined Lua-Terra programs. This process is implemented as a preprocessor that parses the combined Lua-Terra text. This design allows us to implement Terra without having to modify the LuaJIT implementation. The preprocessor parses the text, building an AST for each Terra function. It then replaces the Terra function text with a call to specialize the Terra function in the local environment. This constructor takes as arguments the parsed AST, as well as a Lua closure that captures the local lexical environment. When this code is executed, it will call into an internal library that actually constructs and returns the specialized Terra function. The preprocessed code is then passed to the Lua interpreter to load.

Terra code is compiled when a Terra function is typechecked the first time it is run. We use LLVM [17] 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 front-end 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.

6. Evaluation

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 implementations while achieving high performance. First, we evaluate an auto-tuner for BLAS and a DSL for stencil computations. Next, we show a high-performance class system and container with programmable data layout that can be JIT compiled. Each would be difficult to implement in a single existing language.

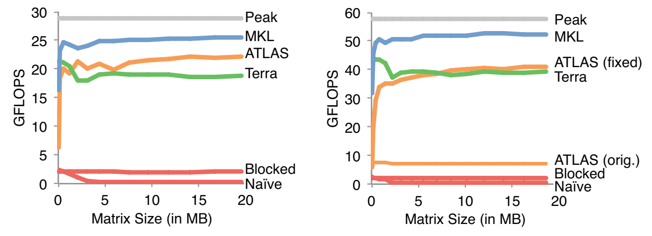
6.1 Tuning DGEMM

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.

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

The ATLAS project [33] 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 [33]. 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 block of the output ma-



(a) DGEMM Performance (b) SGEMM Performance

Figure 6. 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.

trix 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. 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.

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 6 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.

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-

```
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 7. A kernel from a real-time fluid solver written in C (top) compared to Orion (bottom).

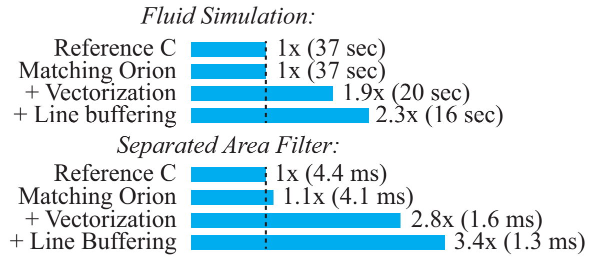


Figure 8. Speedup from choosing different Orion schedules. All results on Intel Core i7-3720QM, 1024x1024 floating point pixels.

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.

6.2 Orion: A Stencil DSL for Images

To test Terra's suitability for DSL development, we created Orion, 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 implemented like the C code in Figure 7, 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 7 shows the same `diffuse` operation written in Orion. Rather than specify loop nests directly, Orion programs are written using image-wide operators. For instance, `f(-1,0) + f(0,1)` adds the 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 [24], a language for the related domain of image processing. The user guides optimization by specifying a *schedule*. An Orion 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, Orion 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 Orion, 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 Orion, we use operator overloading on Lua tables to build Orion expressions. These operators build an intermediate representation (IR) suitable for optimization. The user calls `orion.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 8. Given a schedule that matches the C code, Orion performs similarly, running 10% faster. Enabling vectorization in Orion 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 [26]. 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, Orion 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 Orion. High-level features of Lua made it easy to express transformations on the Orion 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 Orion, Halide, a related image processing language, requires three different languages to provide the same functionality as Orion. It uses C++ for the front-end, ML for manipulating the IR, and LLVM for code generation [24]. From our experience implementing Orion, using Lua to stage Terra code accomplishes the same tasks, but results in a simpler architecture.

6.3 Building reuseable components via type reflection

Type reflection makes it possible to define the behavior and layout of types at a low-level. First, we show the flexibility of Terra's type reflection by using it to implement a class system with subtyping. Then we show how it can be applied specifically to building runtimes for high-performance computing by implementing a type constructor that can automatically generate a data table with either array-of-structs or struct-of-arrays layout.

6.3.1 Class Systems

Using type-reflection, we can implement a single-inheritance class system with multiple subtyping of interfaces similar to Java's. We specify classes using an interface implemented in Lua:

```
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
```

The function `interface` creates a new interface given a table of method names and types. The functions `J.extends` and `J.implements` install metamethods on the `Square` type that will implement the behavior of the class system.

Our implementation, based on vtables, uses the subset of Stroustrup's multiple inheritance [27] that is needed to implement single inheritance with multiple interfaces. For each class, we define a `__finalizelayout` metamethod. This metamethod is called by the Terra typechecker right before a type is examined, allowing it to compute the layout of the type at the latest possible time. For our class system, this metamethod 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 on `class`, we create a stub method to invoke the real method through the class's vtable:

```
for methodname,fn in pairs(concretemethods) do
  local fntype = fn:gettype()
  local params = fntype.parameters:map(symbol)
  local self = params[1]
  class.methods[methodname] =
    terra([params]) : fntyp.returns
      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` metamethod:

```
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.

We measured the overhead of function invocation in our implementation using a micro-benchmark, and found it performed within 1% of analogous C++ code. The implementation requires only 250 lines of Terra code to provide much of the functionality of Java's class system. 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.

6.3.2 Data Layout

Terra's type reflection should help programmers build reusable components in high-performance runtimes. 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, 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 9 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 [19] 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.

7. Related Work

Much work on multi-stage programming has focused on homogeneous meta-programming [22, 29, 30]. MetaML [30] and MetaOCaml [29] add staging annotations to ML. Staged code is lexically scoped, and a type system ensures that the annotations can only produce well-typed programs. MetaHaskell is an extension of Haskell for heterogeneous meta-programming that supports embedding new object languages while ensuring that the staging is type-safe [18]. Unlike Terra, the object languages implemented in MetaHaskell do not share Haskell's lexical environment and are currently unhygienic. Eckhardt et al. propose implicit heterogeneous programming in OCaml with a translation into C [10] but the type language is limited to basic types and arrays. In contrast to statically-typed approaches, Terra supports the creation of user-defined types using arbitrary code but precludes static typing of the full Lua-Terra program.

Heterogeneous multi-stage languages with shared lexical scope and different execution environments have occurred organically in the past [10, 32]. Separating compile-time and runtime-time environments has also been used to make macro expansion composable [11]. To our knowledge, we are the first to argue for these design choices as a way to generate portable high-performance code, retaining interoperability through an optional FFI.

Multi-stage programming has been used to generate high-performance programs [12, 23, 33]. Carette investigates staging of Gaussian elimination in MetaOCaml [5], while Cohen et al. investigate applying MetaOCaml to problems in high-performance computing like loop unrolling/tiling and pipelining [8]. This work has focused on using staging to improve the performance of specific problems. More generally, Chafi et al. use lightweight modular staging [25]—a type-directed staging approach that can be implemented as a library—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 [4, 7]. Additionally, Intel's ArBB enables runtime generation of vector-style code using a combination of operator overloading and macros in C++ [21]. In contrast to Terra, ArBB and Delite do not have explicit staging annotations, instead relying on types to distinguish object-language expressions from meta-language ones. In practice we have found that this type-directed staging makes it difficult to know when code will execute.

The macro systems of Lisp and Scheme have also been used to build DSLs. In particular, Racket [31] 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. Terra, by contrast, is implemented as a separate language from Lua, which allows for different design decisions in each (e.g., automatic memory management in Lua, manual management in Terra).

Previous work examined the combination of staging and type reflection for statically-typed languages. Template meta-programming in C++ is widely used and allows generation and introspection on types. Garcia and Lumsdaine describe a core calculus for compile-time meta-programming based on template meta-programming in C++ [13]. Similar to Terra, their semantics support code generation and type reflection, but like C++ they focus only on ahead-of-time code generation. F# allows type-providers which can specify types and methods based on external data like a SQL schema [28]. Metaphor is a multi-stage language with support for type reflection on a built-in class system [20]. In contrast, Terra's type reflection allows the creation of class-systems as libraries.

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 9. Performance of mesh transformations using different data layouts.

Dynamic languages have added extensions to produce low-level code. Cython is an extension to the Python language that allows the creation of C extensions while writing in Python's syntax [3]. Copperhead supplements Python with a vector-style language that can run on GPUs [6]. In both cases, the low-level code depends on the Python runtime to execute.

Other languages have been proposed as a portable target for low-level code [16, 17]. Terra is also usable directly as a low-level programming language, making it possible to write runtime code in Terra.

8. Discussion and Future Work

We have presented Terra, a staged language embedded in Lua and designed for high-performance computing. By comparing to existing multi-language systems, we have shown that the combination of high- and low-level languages, shared lexical environment, separate execution, and type reflection make designing autotuners, DSLs, and runtime components simpler, while retaining high-performance.

We plan to extend Terra in several ways. Accelerators like GPUs or Intel's MIC architecture provide more performance for data-parallel problems. We plan to extend our implementation so that Terra can generate code that runs on these architectures. Currently Terra does not provide a seamless way to mix Terra code compiled ahead-of-time with dynamically compiled code, which can be problematic for DSLs with a large runtime. We plan to address this with a module system that will allow some code to be generated ahead-of-time, while still allowing JIT compilation of code at runtime.

Terra addresses the problem of generating high-performance code and interoperating with existing applications. We want to generalize the way Terra is embedded and staged to make it easy to embed custom DSLs in Lua in the same way that Terra is embedded. In particular, we think that having DSLs share the same lexical environment during compilation will open up more opportunities for interoperability between different languages. In the future, we envision a programming ecosystem where the right language can be used for a particular task without loss of performance, or significant effort to integrate the language with existing systems.

Acknowledgments This work has been supported by the DOE Office of Science ASCR in the ExMatEx and ExaCT Exascale Co-Design Centers, program manager Karen Pao; DARPA Contract No. HR0011-11-C-0007; and the Stanford Pervasive Parallelism Lab (supported by Oracle, AMD, Intel, and NVIDIA). Any opinions, findings and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA. We thank our reviewers for suggesting improvements to specialization.

References

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

[2] A. Bawden and J. Rees. Syntactic closures. In *LFP*, 1988.

[3] 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.

[4] 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*, 2011.

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

[6] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *PPoPP*, 2011.

[7] 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*, 2011.

[8] A. Cohen, S. Donadio, M. Garzaran, C. Herrmann, and D. Padua. In search of a program generator to implement generic transformations for high-performance computing. In *MetaOCaml Workshop*, 2004.

[9] 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*, 2011.

[10] 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.

[11] M. Flatt. Composable and compilable macros: You want it when? In *ICFP*, 2002.

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

[13] R. Garcia and A. Lumsdaine. Toward foundations for type-reflective metaprogramming. In *GPCE*, 2009.

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

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

[16] S. Jones, T. Nordin, and D. Oliva. C--: A portable assembly language. In *Workshop on Implementing Functional Languages*, 1997.

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

[18] G. Mainland. Explicitly heterogeneous metaprogramming with metahaskell. In *ICFP*, 2012.

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

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

[21] 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*, 2011.

[22] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. C and tcc: A language and compiler for dynamic code generation. *TOPLAS*, 21 (2):1999.

[23] 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):2004.

[24] 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.

[25] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, 2010.

[26] J. Stam. Real-time fluid dynamics for games. In *GDC*, 2003.

[27] B. Stroustrup. Multiple inheritance for C++. In *European Unix Systems Users's Group Conference*, 1987.

[28] 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.

[29] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, 2004.

[30] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, 1999.

[31] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *PLDI*, 2011.

[32] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *ICFP*, 2001.

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