The Design of Terra: Harnessing the best features of high-level and low-level languages

Zachary DeVito[1] and Pat Hanrahan[1]

1 Stanford University
{zdevito,phanrahan}@cs.stanford.edu

Abstract

Applications are often written using a combination of high-level and low-level languages since it allows performance critical parts to be carefully optimized, while other parts can be written more productively. This approach is used in web development, game programming, and in build systems for applications themselves. However, most languages were not designed with interoperability in mind, resulting in glue code and duplicated features that add complexity. We propose a two-language system where both languages were designed to interoperate. Lua is used for our high-level language since it was originally designed with interoperability in mind. We create a new low-level language, Terra, that we designed to interoperate with Lua. It is embedded in Lua, and meta-programmed from it, but has a low level of abstraction suited for writing high-performance code. We discuss important design decisions — compartmentalized runtimes, glue-free interoperation, and meta-programming features — that enable Lua and Terra to be more powerful than the sum of their parts.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases language interoperability, meta-programming, high-performance, Lua

1 Introduction

A common approach for developing large applications is to use multiple languages that are each suited to particular tasks. This approach is used across many areas of development. In web development, high-speed servers (e.g, Apache) used for mostly static web pages are frequently written at a low-level in C/C++ for high throughput. Other parts of the web application might be written in a scripting language such as Javascript (Node.js) or Ruby (Ruby on Rails) for fast prototyping. A similar approach is used in scientific computing where Python, MATLAB, or R may be used for high-level prototyping. Since these languages are often an order-of-magnitude slower than good code in a low-level language, performance critical parts of the application are then re-written in a lower-level language (e.g., NumPy) and accessed using a foreign function interface (FFI). Video games share a similar design with most of the game engine written in a low-level language, and an embedded scripting language (commonly Lua) for describing game events.

Including both a high-level and low-level language is useful because it allows programmers to choose the level of abstraction for each task. Performance critical parts can be coded with an abstraction that is close to the machine which includes features such as manual layout and management of memory as well as access to low-level vector instructions. Other parts can be written more productively in higher-level languages (whether statically-typed like Haskell or dynamically-typed like Lua) that include features such as automatic memory management and higher-level data types which abstract the details of the machine. This design also allows people with different skills to work on the same project. A game designer might only need to understand Lua to be productive.

The approach of using multiple languages is also widely used in software development itself. A scripting language such as a shell script or Makefile control the compilation and deployment of software written in another language. This usage differs from the others because the use of multiple languages occurs during compilation and is used to organize or generate code. Nevertheless, multiple languages are still involved and must interact.

Most programming languages being used this way, however, were not designed with multi-language interoperability in mind. In applications that use multiple languages, this often manifests in two ways: as extra work in "gluing" the two languages together, and extra complexity in each language in the form of duplicated features.

Glue code can appear for many reasons. Symbols in one language need to be bound in the other. Data passed from one language to the other might need to be manually transformed. If values allocated in the runtime of one language can be seen from the other, then glue may be needed to ensure correct object lifetimes. For instance, C extensions to Python use `Py_INCREF` and `Py_DECREF` to maintain handles to Python objects and ensure they are not garbage collected.

Complexity also arises because multiple languages are often solving the same problems. Each might have its own solution for namespaces, package management, and compilation. This can produce a system that contains the worst aspects of each language. For instance, an application written in a scripting language does not require a separate compilation step, making iterative coding easier at the cost of static checking. But when a compiled language is added, the application now requires a compilation step *and* does not do any static checking on the scripting language.

We investigate an improvement to two-language programs that uses languages that were designed with this kind of interoperability in mind. For our high-level language, we use Lua since it was designed to interoperate with low-level C code [11]. C was not designed with interoperability in mind so instead we replace it with a new low-level language, Terra, that we designed from the beginning to interoperate with Lua. Terra's textual representation is embedded in Lua code. It is meta-programmed from Lua, and it relies on Lua to describe its static types. But Terra itself exposes a low-level of abstraction suited to writing high-performance code.

The approach of using two languages designed to interoperate allows each language to remain simple relative to single-language solutions, but the interaction between both languages allows for behavior that is more powerful than the sum of their parts. We can eliminate the glue code that couples two languages together and ensure that the languages do not contain redundant solutions for common problems.

In this paper we discuss important decisions we made when designing Terra to enable two-language interaction while keeping each language simple:

1. We compartmentalize the runtimes of Lua and Terra, allowing each language to focus only on features that they are good at. Terra code can execute independently from the Lua runtime, ensuring a low-level control of performance. It can also run where Lua cannot, such as on GPUs.
2. Interoperability does not require glue code since Terra type and function declarations are first-class Lua statements, providing bindings between the two languages automatically.
3. Lua is used as the meta-programming language for Terra. We provide facilities for generating code and types dynamically, removing the need for a separate offline compiler. We simplify Terra by offloading typical features of a compiled low-level language such as class systems to the Lua meta-program. It also enables powerful behavior such as runtime generation of low-level code.

We start with background on programming in Lua and Terra, examine these design decisions in more detail, and briefly discuss our experiences using the language.

2 Lua and Terra by Example

Terra code itself is syntactically embedded in a Lua program. The keyword `terra` introduces a Terra function (`function` is used for Lua). Here is an example that declares both a Lua function and a Terra function:

```
  function addLua(a,b) -- Lua function
      return a + b
  end
  terra addTerra(a : int, b : int) : int  -- Terra function
5     return a + b
  end
  print(addTerra(1,1)) -- 2
```

Both share a similar syntax, but have different semantics. Lua is a dynamically-typed language with semantics that are similar to Javascript, including automatic memory management and high-level data types based on associative tables. In contrast, Terra is a low-level language with semantics that are analogous to C. It has pointer arithmetic, manual memory management, and low-level data types like C's. Terra functions include type annotations and are statically typed in the sense that types are checked at compile time. Compilation of Terra itself occurs dynamically as the Lua program executes.

Lua code can call Terra functions directly, as illustrated on line 7. This call causes the Terra function to be compiled to machine code. The arguments are converted from Lua values to Terra values, and the result is converted back to a Lua value (Section 3.2 has more details about this process).

Terra types are also declared using statements in Lua:

```
struct FloatArray {
    data : &float
    N : int
}
-- method declaration:
terra FloatArray:get(i : int) : float
    return self.data[i]
end
```

Terra entities (e.g., types, functions, expressions, symbols) are all first class values in Lua. They can be held in Lua variables and passed through Lua functions. For instance, we can get a result similar to C++ templating by simply nesting our type and function definitions inside a Lua function that takes a type as its argument:

```
function Array(ElemType)
  local struct ArrayType {
    data : &ElemType
    N : int
  }
  terra ArrayType:get(i : int) : ElemType
    return self.data[i]
  end
  return ArrayType
end
```

Both code (the method `get`) and types (the definition of `ArrayType`) can be defined dynamically in this way. When defined they capture the values in the local environment that they reference such as `ElemType`. We also allow the generation of arbitrary code using an approach based on multi-stage programming [5], and the generation of arbitrary types using a customizable API that runs during typechecking based on meta-object protocols [6]. We describe this functionality in more detail in Section 3.3.

3 Design Decisions

Our goal when designing Terra was to make a low-level language that complements the design of Lua and supports two-language programming as used in practice. Other approaches to two-language interoperability have focused on retrofitting existing languages [2, 7, 9, 18], and while these approaches can reduce glue code, the combination of both languages ends up more complicated than if the languages themselves were originally designed with interoperability in mind. Lua's design values simplicity as its most important aspect [10], so it is important that Terra is simple as well. Our design decisions focus on providing the benefits of multi-language programming while minimizing the complexity added to either language to support it.

3.1 Compartmentalized Runtimes

Though Terra programs are embedded inside Lua, we compartmentalize the runtimes of Lua and Terra. Each can run independently from the other, and only interact through fixed channels that we describe in the next section. This is one reason we refer to Terra as a separate language from Lua rather than an extension to it. This design decision allows both Lua and Terra to thrive for the tasks they are suited towards. We limit interaction to functionality that does not impose additional complexity on either runtime.

Each language has its own separate approach to code execution and memory management. For Lua, we use LuaJIT [2], a just-in-time trace-based compiler for Lua code. It handles the dynamic features of the Lua language well. Lua has its own heap where its objects are managed using garbage collection. In contrast, Terra code goes through a different pathway. It is compiled directly to machine code and optimized using LLVM [14]. This approach is less suited to dynamic features and takes longer than JIT compilation, but produces high quality low-level code. To further control the performance of Terra, memory is manually managed with `malloc` and `free`.

Alternative designs that combine high- and low-level programming attempt to augment higher-level languages with constructs that express lower-level ideas. For instance, in certain dialects of LISP, annotations allow the compiler to elide dynamic type safety checks to get higher numeric performance [20, 25]. Other dynamically-typed languages allow the optional annotation of types [3, 4, 7, 19, 27]. For instance, Cython [3] allows the programmer to optionally tag certain types in a Python program with C types, replacing them with a higher-performance implementation. Other types, however, go through the normal Python API. Type annotation can help eliminate boxing of certain values which often helps improve performance in local areas.

However, there is a difference between improving performance of a high-level program and obtaining near optimal performance that is possible when programming at a low level. Near optimal performance is often a composition of global factors including the specific memory layout of objects, careful instruction selection for inner loops, and use of advanced features such as vector instructions.

Intermixing high-level features with low-level features makes this careful composition harder to control. For instance, a missed annotation on a higher-level type may cause insertion of guards that will ruin the performance of an inner loop. The complexity of this approach makes the result harder to debug compared to a language like C where an expert programmer can disassemble the code and work out what is going on. In our experience building high-performance domain-specific languages [5, 8], the last order-of-magnitude of performance comes from this process of understanding the assembly and tweaking the low-level code that produces it. Furthermore, when high-performance facilities are only optionally offered, there is a tendency for libraries to be written without them, forcing anyone looking for whole-program performance to reimplement the library at a low level.

Maintaining compartmentalized runtimes has additional advantages for low-level programming. Since the runtimes are not intertwined, it is possible to run Terra code in places where the high-level Lua runtime does not exist. We can run Terra code in another thread (we use `pthreads` directly) without worrying about interaction with Lua's runtime or garbage collector. We can save Terra code generated in one process for offline use (we provide a built-in Lua function `terralib.saveobj` to do this). This allows us to use Terra as a traditional ahead-of-time compiler while still getting the benefits of meta-programming from Lua. We can also use Terra code where Lua cannot run, such as on NVIDIA GPUs. A built-in function `terralib.cudacompile` takes Terra code and produces CUDA kernels.

Finally, compartmentalizing the runtimes means that the semantics of each runtime is very similar to existing languages (Lua and C), each of which have benefitted from a large amount of engineering effort and optimization. Since we do not modify the semantics of these runtimes drastically, we can reuse this effort. In our case, this means using LuaJIT and LLVM as libraries for Lua and Terra, respectively. Terra itself is only around 15k lines of code (compared to Lua's 20k) and substantially shorter than languages whose semantics do not closely match C/LLVM such as Rust (> 110k lines). Furthermore, Terra's similarity in types and semantics to C makes backwards compatibility possible. We include a library function `terralib.includec` that can parse C files and generate Terra bindings to C code.

3.2 Interoperability without Glue

Though the runtimes are compartmentalized, we still provide ways for Lua and Terra to interact. Terra code is embedded in Lua as a first-class value. We chose a Lua-like syntax for Terra as opposed to an alterative, such as using C's syntax, because it made parsing the extension easier and gave the languages a consistent interface. This is in contrast to languages frameworks like Jeannie [9] or PyCUDA [13], which preserve the syntax of an original language.

Both languages also share the same lexical environment as well. A symbol `x` in scope in Lua code is visible in Terra code and vice versa. We saw this feature in the example code where Terra code refers to `ElemType`, a Lua variable representing a Terra type (in fact, all types such as `int` or `double` are exposed in this way). When a Terra function references another function when making a function call, the reference is resolved through Lua's native environment during typechecking.

Sharing a lexical environment might seem surprising given that the languages have compartmentalized runtimes, but one way to think of this design is that the Terra *compiler* is part of the Lua runtime. Terra code itself runs separately from its compiler, much like the design of other compiled languages. It turns out sharing this scope simplifies the process of defining Terra code. Languages like C require their own syntax for declarations and definitions whereas Terra simply tracks these features as values in the Lua state. As we saw in Section 2, powerful features such as templating simply fall out of this decision. Additionally features like namespaces are accomplished by storing Terra functions in Lua tables and conditional compilation is done through Lua control flow. This design simplifies the implementation of Terra's compiler as well, since its symbol tables are simply the already present Lua environment.

We also want to be able to use values from one language in the other. We take an approach similar to other interoperability frameworks such as those used for Scheme and C [18], or LuaJIT and C [2]. When necessary, such as across functions calls between languages, values from one language are transformed using a set of built-in rules to values in the other. Numeric types often translate one-to-one, while aggregate types have more sophisticated rules. For instance, a Lua table `{real = 4, imaginary = 5}` can be automatically converted to a Terra struct with definition `struct Complex { read : float, imaginary : float}`.

It is also useful to allow one language to directly view a value in the other, rather than perform a transformation which necessitates a copy. This interaction, however, can become complicated when the low-level language can view high-level data structures that are managed via garbage collection since it must alert the runtime that a reference remains to the object. It would also compromise the compartmentalization of the runtimes, since it encourages Terra code to use Lua objects directly, which involves making Lua runtime calls inside Terra code. We adopt Lua's "eye-of-the-needle" approach [11] to this problem, which prevents Terra code from directly referencing Lua values. Instead Terra code can manually extract primitive data values from the Lua state using Lua API functions, or it can call back into Lua code directly.

The opposite behavior, viewing manually-allocated objects from a garbage collected language, has different constraints. These objects are already being managed by hand and performance considerations when accessing them from Lua are not as critical. For this direction, we adopt the approach of LuaJIT's FFI, which allows Lua to introspect aggregate structures like Terra's structs. A Terra struct `x` of type `Complex` can be returned to Lua without conversion. Statements in Lua such as `x.real` will dynamically extract the appropriate field and apply the standard conversion rules. For simplicity, Terra objects referenced from Lua follow the same rules for memory management as if they were used in Terra directly. That is, the programmer is responsible for ensuring a Terra object referenced from Lua is not freed too early, and for freeing it appropriately. This behavior makes interaction between the two languages possible without adding major complexity to either language.

A difference between our approach and the other frameworks is that the descriptions of Terra types are already natively declared as Lua values so it is not necessary to provide separate descriptions of aggregate datatypes through a side-channel. Other approaches that integrate with C (for instance) either need to parse the header files to see the declarations, or have a separate way of manually declaring the low-level types. We reduce redundancy and the complexity of the language by making Terra type definitions a part of the Lua runtime.

3.3 Meta-programmed by Default

Finally, we use Lua as the meta-programming language for Terra. This is useful for several reasons. First, it reflects a common use of scripting languages where a shell script or Makefile is used to arrange the compilation of low-level code. Making this process more seamless enables more powerful transformations, including building auto-tuned libraries or entire domain-specific languages that compile to Terra. Secondly, relying on Lua for meta-programming allows us to keep the core of Terra simple without limiting the flexibility of the language.

Terra provides facilities for both dynamically generating code, and for dynamically generating types. For code, we use a version of multi-stage programming [24] with explicit operators. A *quotation* operator (the backtick `) creates a fragment of Terra code, analogous to creating a string literal except that it creates a first-class value representing a Terra expression rather than a string. An *escape* (e.g., [lua_exp]) appears inside Terra code. When a Terra quote or function definition is encountered during the execution of the Lua program, the escaped expression lua_exp will be evaluated (normally to another quote) and spliced into the Terra code. It is analogous to string interpolation. For instance consider the following simple program:

```
function gen_square(x)
    return `x * x
end

terra square_error(a : float, b : float)
    return [ gen_square(a) ] - [ gen_square(b) ]
end
```

When square_error is defined, the gen_square calls in the escapes will evaluate, generating quotes that square a number, resulting in a function whose body computes a * a - b * b.

Generation of Terra code, like that shown above, is similar to other meta-programming frameworks such as LISP macros, or traditional multi-stage programming [16, 24]. But Terra also includes the ability to generate types dynamically as well. Terra is statically typed in the sense that types are known at compilation time, but Terra functions are compiled during the execution of the Lua program, which gives us the ability to meta-program their behavior. This is a different design from statically-typed multi-stage programming where the behavior of all types (those in the first stage and later stages) are checked statically. While it gives up the advantage of type checking the program entirely ahead-of-time, it also adds a lot of flexibility. For instance, we allow objects to programmatically decide their layout and behavior.

As an example, we may want to write a program that reads a database schema and generates a type from that schema, such as this student type:

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

Rather than hard-code the type, we can construct its behavior programmatically:

```
Student = terralib.types.newstruct() -- create a new blank struct
Student.metamethods.__getentries = function()
    -- Read database schema and generate layout of Student.
    -- (e.g., { {field="name", type=rawstring},
    --          {field="year", type=int}})
   return generated_layout
end
```

When the typechecker sees a value of Student, it asks for its memory layout by calling the __getentries function defined for the type, which returns a data structure describing the layout. The layout is defined before compilation, allowing the generation of high-performance code, but the implementation of __getentries can adapt to runtime information. The behavior of methods is also determined programmatically, allowing the generation of “setter” methods automatically:

```
   Student.metamethods.__getmethod = function(self,methodname)
       local field = string.match(name,"set(.*)")
       if field then
         local T = Student:typeoffield(field)
5        return terra(self : &Student, v : T)
           self.[field] = v
         end
       end
       error("unknown method: "..name)
10 end)
```

When the typechecker sees that `setname` is not explicitly defined, it will call the user-provided Lua function `__getmethod` to create it. We use string matching (line 2) to find the field name, and generate a body for the method (lines 5–7).

Terra’s type system is a form of meta-object protocol [12]. It is similar in some ways to F#’s type provider interface [23], but describes types at a low-level like C rather than in a runtime such as .NET’s common language runtime (CLR). It combines features of meta-object protocols from dynamic languages with the generation of low-level statically-typed code, and is described in more detail in [6]. Furthermore, this mechanism is the only form of user-defined type built-in to Terra, apart from some syntax sugar for common cases such as defining a statically-dispatched method.

Using a full scripting language to meta-program code and types allows us to omit many features of typical statically-typed language, keeping Terra simple. While this approach appears different from other statically-typed languages, it is possible to think of *every* statically-typed language as a meta-program containing multiple languages. One language, at the top level, instantiates entities like types, function definitions, classes, and methods. Another language in the body of functions defines the runtime behavior. The top-level language “evaluates” when the compiler for that language is run producing a second stage containing the actual program. But this language at the top level is normally not a full programming language. Some languages may have more built-in features (e.g. inheritance, traits, etc.) but these are very specific to the language. Having too few features in this top-level language, such as the initial lack of generics in Java, can prevent a language from being sufficiently expressive and provides little ability to grow the language [21], causing people to resort to ad-hoc solutions like preprocessors or source-to-source code generators to produce generic code.

Our approach in Terra replaces this limited top-level language entirely with Lua. Doing so removes the need for many language-specific features but provides a powerful mechanism to grow the language. In Terra features like class systems or inheritance are implemented as libraries that meta-program types. This behavior is familiar in dynamically-typed languages such as LISP’s CLOS [12], but we extend it to work with a low-level compiled language.

4 Alternatives

Terra and Lua represent one possible design for a two-language system. It is also worth considering alternatives. We chose Lua as our high-level language due to its simplicity and its support for embedding in other languages, but most of the design decisions could be easily adapted to other dynamically-typed languages as well such as Python, Ruby, or Javascript. Another possibility is to use a statically-typed language such as Haskell or Scala as the high-level language. Systems such as lightweight modular staging in Scala [17] or

MetaHaskell [15] are examples of multi-language systems that take this approach. Their primary advantage is the increased safety of the high-level language. For example, some systems such as MetaHaskell or traditional multi-stage programming in MetaML [24] can statically guarantee that any code that can be generated in the target language will be well typed. Other systems such as the C++ library for generating LLVM [14] do not statically guarantee the type correctness of generated code, but still benefit from being able to give static types to the target language's core concepts including its functions, types, and expressions.

However, using a statically-typed high-level language in a two-language system can introduce added complexity. It adds an additional set of typing rules to the high-level language that is different from the low-level type system, as well as an additional phase of type checking. Some meta-programming systems such as MetaML avoid this complexity by making the staging language and target language the same, but this design is not possible when we explicitly want to use different languages.

Furthermore, the addition of static types can introduce complexity in other areas of the design, such as cross language interoperability. Static type systems often include some way of modeling abstract data types and behavior associated with them such as type classes or object-oriented class systems. Support for interoperability becomes more complex if it needs to support these built-in class systems. This complexity is evident in runtimes such as .NET's CLR. Calling from CLR managed code to unmanaged C/C++ code requires the generation of wrappers by the runtime so that CLR objects can be viewed from unmanaged code and vice-versa. It has several pathways to support different use cases (P/Invoke, COM wrappers, C++ wrappers) [1]. The added features to support class system behavior in such a model make it harder to compartmentalize the two runtimes and more difficult to understand their interaction.

Our goal creating Terra was to enable interaction between two languages while keeping the entire system as simple as possible. We chose to omit static typing in the higher-level language to decrease the complexity of the system as a whole. For example, since class systems are not built into Lua as they are in the CLR, the semantics for translating values from one language to another are simpler.

5 Experiences

We have used Terra to implement many applications in areas including linear algebra libraries, image processing, physical simulation, probabilistic computing, class system design, serialization, dynamic assembly, and automatic differentiation [5, 6, 8]. Our experiences using the system suggest it allows for simpler solutions by eliminating the glue code and management of separate build systems that occur in other approaches.

We found that many design patterns that occur in statically-typed languages can be expressed concisely in Terra, with performance that is similar to other low-level code. We implemented class systems similar to Java's (250 lines) or Google's Go language (80 lines) [5]. We were also able to implement concise generic proxy classes, such as an `Array(T)` type that forwards methods to its elements in only a few lines of code [6]. It performs as well as proxies written by hand in C. Patterns requiring introspection on types, such as object serialization were particularly well suited to Terra's design. For instance, we created a simple but generic serialization library in about 200 lines that could serialize scene graphs at 7.0GB/s [6] (compared to 430MB/s for Java's Kryo [22] library). The integrated solution to meta-programming allowed us to consider more aggressive optimizations that would normally only be done in separate preprocessors. A dynamic assembler we created [6] automatically generated method implementations to assemble instructions using a simple pattern language. It could also fuse patterns together into templates, resulting in speeds 3–20 times faster than the assembler used in Google Chrome.

For software programs that rely heavily on scripting to build libraries, such as autotuners, we were able to provide much simpler solutions. Our autotuner of matrix multiply can create code that runs within 20% of the ATLAS [26] autotuner and requires only 200 lines of code (compared to thousands in ATLAS) [5]. This simplicity arises from eliminating a lot of duplicate technologies. ATLAS uses `Makefiles`, preprocessors, offline compilers, and shell scripting. Combining this functionality into a single two-language system removes most of the complexity of the build systems, allowing the programmer to focus on code optimization.

Finally, the generic meta-programming features in a low-level language make it useful in developing high-performance domain-specific languages. We have used it to implement languages for probabilistic programming [5], and image processing [8] that are competitive with solutions written using traditional software tools but are easier to architect and distribute. Currently the use of domain-specific languages is not widespread. One reason for this is that current build tools do not make it easy to generate low-level code generically. We think that using this integrated two-language design as the basis for more software systems will make it easier to design, develop, and integrate high-level domain-specific languages in practice, allowing the use of higher-level abstractions without sacrificing overall performance.

References

1 Interoperability overview (C# programming guide). `https://msdn.microsoft.com/en-us/library/ms173185.aspx`.

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

3 Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science and Engineering*, 13.2:31–39, 2011.

4 Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, 2012.

5 Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A multi-stage language for high-performance computing. In PLDI, pages 105–116, 2013.

6 Zachary DeVito, Daniel Ritchie, Matt Fisher, Alex Aiken, and Pat Hanrahan. First-class runtime generation of high-performance types using exotypes. In PLDI, pages 77–88, 2014.

7 Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In OOPSLA, pages 231–245, 2005.

8 James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. In SIGGRAPH, pages 144:1–144:11, 2014.

9 Martin Hirzel and Robert Grimm. Jeannie: Granting Java native interface developers their wishes. In OOPSLA, pages 19–38, 2007.

10 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In HOPL III, pages 2:1–2:26, 2007.

11 Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. Passing a language through the eye of a needle. *Commun. ACM*, 54(7):38–43, July 2011.

12 Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.

13 Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.*, 38(3):157–174, 2012.

14 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In CGO, 2004.

15 Geoffrey Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In ICFP, pages 311–322, 2012.

16 John McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, August 1978.

17 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In GPCE, pages 127–136, 2010.

18 John R. Rose and Hans Muller. Integrating the Scheme and C languages. In LFP, pages 247–259, 1992.

19 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

20 Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. Technical report, 1999.

21 Guy L. Steele, Jr. Growing a language. In OOPSLA Addendum, pages 0.01–A1, 1998.

22 Nathan Sweet. Kryo. `https://code.google.com/p/kryo/`.

23 Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. F#3.0 — strongly-typed language support for internet-scale information sources. Technical report, 2012.

24 Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, pages 203–217. ACM Press, 1999.

25 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In PLDI, pages 132–141, 2011.

26 R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Softw. Pract. Exper.*, 35(2):101–121, February 2005.

27 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In POPL, pages 377–388, 2010.