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

Jan Seeger

23.1.2014

Abstract

Writing high-performance code is a work-intensive process. Often, performance-optimized code is difficult to maintain, difficult to modify, and dependent on architecture specific properties for its performance. Using code generation can ease this burden. By using a multi-stage language to generate code, optimizations can be stated in a more abstract fashion. In this paper, we will describe the multi-stage programming language Terra and how it supports generating high-performance code without sacrificing readability or generality.

1 Introduction

Writing programs is hard. Writing programs that work correctly, and also satisfy non-functional requirements, such as performance, is even harder. When the performance requirements of a program increase, more and more optimizations are required in order to satisfy those requirements. This makes highly optimized program code hard to read and understand. By using program generators, high-performance code can be written that does not suffer under the complexity of performance optimization a program generator also allows performance optimizations to be stated in a generic way, easing porting of generated code across platforms.

However, generating high-performance programs is often done in an ad-hoc way. Software that uses program generation techniques for creating high-performance implementations (such as ATLAS[28]) often relies on a heterogeneous set of tools (such as configure scripts, the GNU automake tools or similar). Software implemented in C or C++ often uses techniques that rely on the preprocessor (such as the X-Macro approach[16]) or the C++ template processor (template metaprogramming [26]) for generating code. All these approaches have in common that the language used to generate code is different from that of the code itself.

With multi-stage languages, the language used to write code generators is the same as (or very similar to) the code itself. This reduces the potential for error and allows a programmer to switch freely between program and generator. It reduces the difficulty of writing program generators, and frees the programmer from having to learn yet another language for writing program generators.

In order to describe the advantages of using a multi-stage language for writing program generators, we will introduce the language Terra, a multi-stage language for generating high-performance code. To this end, we will show the facilities that Terra makes available to the programmer to write multi-staged code, and present example implementations of several small programs. We will shortly describe the implementation of Terra, and the effects on the language. We will discuss some performance applications that can be implemented in Terra using multi-stage techniques. We will describe some other multi-staged languages and applications their differences to Terra briefly. Finally, we will end with an outlook on the adoption and usage of heterogeneous multi-stage languages such as Terra.

2 Motivation

Consider a toy example like implementing the `sin()` function for an embedded system. If the architecture does not provide a sine function, often the function is implemented using a lookup table. This table contains precalculated values for a range of function arguments. However, this lookup table needs to be created in some way. Manual generation is error-prone and labor-intensive, so often, external scripts are used to generate the table. For example, our problem could be solved by a Python program that generates the lookup table and writes it to a header file. This approach is arguably less arduous than manually writing the table, but the probability of errors is still quite high. There are absolutely no guarantees that source code generated in this way will be correct, both syntactically and semantically. In contrast, consider the example Terra program that implements sine calculation using a lookup table in listing 1. The code is short, and the languages used in the example are very similar to each other. Even in this simple example, program generation can reduce the programmer's burden in writing correct code, and generate performant programs without having to interact with several languages.

```
local sines = {}
for i = 0,179 do
  sines[i] = math.sin(math.rad(i))
end
local lookup = terralib.new(double[180], sines)

terra sin(rad: int) : {double}
  return lookup[rad % 180]
end
```

Listing 1: Precomputing a lookup table with Terra

Program generators are not a new technology by any means, and have been widely adopted in some areas. However, commonly used program generators are often either special purpose, or use a very low-level representation of the manipulated program. Examples for special-purpose program generators are code generators integrated into IDEs (such as refactoring wizards or interface builders) and parser generators (such as yacc, or bison). While these special-purpose tools excel at the tasks they were designed for (such as building graphical user interfaces, or writing boilerplate code), they have no applications outside those specific areas. On the other hand, general code generation tools such as GNU autogen[7] or Cog[1] often work with program code in a language-agnostic way, manipulating programs as string fragments that are glued together by string interpolation or concatenation. This does not offer a big improvement over manual generation of code, since programs generated by string operations can still be erroneous, and the generation tools can not offer any support in finding and correcting errors.

Wouldn't it be much nicer to be able to use the language we already have to generate the code we want, without having to use yet another program, and without having to fear syntax and type errors? This is what multi-stage languages aim to provide: They offer integrated facilities for program generation and also guarantee the absence of errors detectable at compile time in the generated program. Integrated support for program generation makes writing program generators easier and more convenient, and tighter integration into the languages makes generated code less error prone.

3 Concepts of Multi-Stage Programming

In order to describe the multi-stage features of Terra, some basic concepts of multi-stage programming need to be introduced. The word "stage" in multi-stage programming refers to distinct phases in the execution of a program, that differ either by frequency of execution or availablility of information (compare [12]). Stages can be defined as coarsely or finely as the implementor wishes. It is easy to see that taking advantage of this staging information allows more efficient programs to be generated. As an example, consider the (coarse) stages of a compiled program. A compiled program can be said to have two stages, compilation and execution. Preprocessor directives are run at compile time, while regular program code is run at runtime. Using the preprocessor, it is possible to do less work at runtime by generating code at compile-time, and thus generate a higher-performance program. This is a technique that is also used when programming in multi-staged languages, but we will see in chapter 5 that MSP languages allow the transparent implementation of a host of other optimizations. In general, multi-stage programming allows the programmer to take advantage of the staged structure of programs and thus to generate more general as well as better performing code.

As multi-stage programming languages allow the generation of other programs, they also fall in the field of *meta-programming*. Meta-programming is

the study of programs generating other programs. For an overview of metaprogramming techniques and implementations, see [21].
When we run a multi-stage program, in reality we are executing more than one program: Rather, we are running a number of programs equal to the number of stages in the multi-stage program. Each program in one stage generates the program in the next stage. In metaprogramming parlance, the program that is executed and that manipulates other programs is the *metaprogram*, while the program being manipulated is called the *object program.* So each stage (except the first and the last) in a multi-stage program is a metaprogram as well as an object program.
In order to describe the implementation of these concepts in Terra, we will shortly introduce the language syntax in the next chapter.

4 The Terra language

Terra is a multi-staged language created by DeVito et al. in order to create a language suited for high-performance program generation problems[6].
Strictly speaking, Terra consists of two languages: The metalanguage Lua (see `lua.org`), and the object language Terra. When speaking of the Terra language (including the lower-level Terra languages as well as the upper level Lua language) as a whole, we will use "the Terra MSP language" from now on, while the lower level language embedded in the Terra MSP language will simply be called "Terra". Terra is a low-level language comparable to C, while Lua is an embeddable and extensible scripting language with dynamic typing and automatic memory management.

4.1 Syntax

In order to explain the features of the Terra MSP language, we will briefly describe the syntax of the Terra language. We will only briefly present the features of Lua that are interesting for multi-stage purposes. For an in-depth discussion of the Lua language, see [11].
Lua has several features that make it suited as a metalanguage: Its focus on embeddability and extensibility means that extensions to Lua are easy to design and implement, while its conventional and clean syntax presents low hurdles of entry. The extensibility of Lua makes it easy to implement new language features, without having to greatly modify the interpreter. A particular point of interest for writing multi-staged programs is support for first-order functions. The Terra MSP language supports first-order functions in Lua for both Lua and Terra functions. This greatly simplifies handling of Terra functions in the top-level language.
Terra is the lower-stage component of the Terra MSP language. It is a low-level language, similar to C. In fact, Terra is backwards compatible to C, and can import and use functions from C header files. As its closeness to C implies, Terra supports pointers, function pointers and arrays in much the same way as C. Memory management must also be done manually. For an example of Terra code, see listing 2.

```
str = terralib.includec("string.h")
ctype = terralib.includec("ctype.h")
io = terralib.includec("stdio.h")

-- Note type declaration
terra upcase_string(s: &int8): &int8
-- Variables need declaration
-- as well as type declaration
  var len: int = str.strlen(s)
  for i = 0, len do
    s[i] = ctype.toupper(s[i])
  end
  return s
end

io.printf("%s\n",
       upcase_string("bliblablub"))
```

Listing 2: String operations in Terra

This code looks like a C program, with some syntactic differences: The syntax of Terra is strongly influenced by Lua. Some differences to Lua are type specifications (similar to C, but can sometimes be elided), variable declarations with the `var` keyword and operations such address-of (`&`) and dereferencing (`@`). While the syntax of Terra is strongly influenced by Lua, the absence of several features of Lua is notable: First class functions are not supported, as well as the associative array (or "table") so important to Lua.
However, Terra offers some syntactic sugar that is not present in C and simplifies implementing code that mimics object-oriented programming. Such features are multiple return values, automatic pointer dereferencing and a special syntax for defin-

ing methods of structs. For an example of these techniques, see the implementation of a simple stack in listing 3.

```
cio = terralib.includec("stdio.h")

-- Struct definition similar to C
struct StackElem {val: int, next: &StackElem};
struct Stack {height: int, elems: &StackElem};

terra newStack() : {&Stack}
  var newstack: &Stack =
    [&Stack](std.malloc(sizeof(Stack)))
-- Automatic pointer dereferencing
  newstack.height = 0
  newstack.elems = nil
  return newstack
end

-- Define a method for the Stack
terra Stack:push(val: int) : {}
-- function body omitted
end

-- Multiple return values
terra Stack:pop() : {int, rawstring}
-- function body omitted
end

local s = newStack()
-- Method call syntax
s:push(3)
--Access multiple return values
local ret, err = s:pop()
```

Listing 3: Structs, methods and multiple return values in Terra

While these devices make programming in Terra relatively easy and convenient, the true power of the Terra MSP language comes from the fact that it is run and parametrized by the upper stage Lua code. In the next chapter, we will describe the interaction between higher-level Lua and lower-level Terra code and show some examples of the expressive power made available by this combination.

4.2 Multi-staging in Terra

Recall that Terra consists of two languages: Lua and Terra. Lua is the higher-level language, and controls execution and generation of Terra code. This makes the MSP language Terra a *heterogeneous* multi-stage language, a language where the object language is different from the metalanguage. This has the advantage that meta- and object language can be selected so that the resulting MSP language has some desired properties. Terra consists of Lua, which is a flexible, dynamically typed upper-level language, and Terra, a lower-level language which is statically typed. This allows flexibility when generating code, and generation of high-quality code for the lower level language.

However, a drawback of heterogeneous multi-stage programming languages is that they often do not support more than two stages. Terra only supports a maximum of two stages, while other MSP languages (such as MetaML) support an arbitrary number of stages.

As we stated in chapter 4, the power of Terra stems from the fact that evaluation of Terra code is done under control of high-level Lua code. In order for Lua code to control evaluation and parametrization of Terra code, we need facilities to generate and execute Terra program code in Lua. The Terra MSP language uses a quasiquotation-like approach that occurs in similar form in other languages (such as Haskell[14] or Lisp[2]). The Terra MSP language uses *quotes* and *splices*: Quotes are used in Lua code in order to generate Terra expressions, and splices are used in Terra code in order to evaluate Lua expressions and insert their value.

Quotes in Terra can be either the character ` for expressions (for example `` `5 * 3 `` denotes a Terra expression that evaluates to) 15) or the `quote` statement for quotes that contain statements (like the `quote` statement in listing 4).

Splicing is done by using the square brackets `[]`. When Lua code within square brackets is encountered in Terra, the Lua code is executed and the result is placed where the splice was encountered.

Together, these two constructs allow the generation of arbitrary code. Also note that `` [`expr] `` is equivalent to writing `expr` directly in Terra code. For a closer look at the interaction between Terra and Lua using splices and quotes, see listing 4. The code in listing 4 generates 10 power functions, each with an unrolled loop containing exactly the right number of multiplications. The code is generated by the Lua function `generate_power` by returning quotes containing the necessary number of multiplications.

```
function generate_power(x, y, z)
  local stmts = terralib.newlist()
  for j = 1, z do
-- Quote Terra code
    stmts:insert(quote
                x = x * y
              end)
  end
  return stmts
end


mymath = {}
for i = 1,10 do
  mymath["pow" .. tostring(i)] =
    terra(a: uint64) : uint64
      var r:uint64 = 1;
-- Splice code returned by generate_power
      [ generate_power(r, a, i) ]
      return r
    end
end
```

Listing 4: Generating staged power functions

Beyond these explicit quotes and interpolations, values are also automatically transported between Terra and Lua. Terra functions share the lexical environment of the Lua code. This means that variables are not only looked up in the scope of the Terra function currently being compiled, but also in the surrounding Lua code. This means that explicit interpolation is often not required for parametrizing Terra code. Listing 2 already makes use of these features: all code outside the `terra` function declaration is Lua code. `str`, `ctype` and `io` are Lua tables that are filled with C functions. In the Terra function, `str.strlen()` accesses the `strlen` key of the Lua variable `str`. Since tables can contain Terra function and types, this makes it possible to use Lua tables as a simple module system for Terra.

For a diagram of the control flow between Terra and Lua, see figure 1. Execution begins by running Lua code, and the multi-staging constructs of the Terra MSP language switch between generating Terra code, and executing Lua code. When a quote is encountered, Terra code is generated, and when a splice is encountered in Terra code, Lua code is executed. Finally, when a Terra function is called, the generated Terra code is executed.

Note the difference between variables `x`, `y`, and `z` in the call to `generate_power` in listing 4: While `z` is a Lua variable (and thus is known when `generate_power` is executed), `x` and `y` are Terra variables. Terra variables do not yet have a value when the `generate_power` function is called. Therefore, Terra variables evaluate to a reference to themselves when used in Lua. Also note

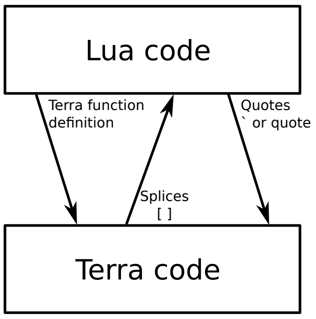

Figure 1: Interaction between Terra and Lua

that in order to prevent unintended variable capture, Terra variables in quotes are renamed: When running the program in listing 5, the variable `x` from `returnx` is renamed to `$1`. This prevents unintentionally reusing variable names, which may cause programming errors.

```
function returnx()
  return quote
    var x = 3
  end
end

function splice()
  return terra()
    var x = 5
    [ returnx() ]
    print(x)
  end
end

splice():printpretty()
```

Listing 5: Variable renaming in Terra

The Terra MSP language is not limited to splicing only expressions and statements: All Terra types are Lua values. For example, in the declaration `var i: int = 0;`, `int` is a Lua value that can be inspected in Lua code. Information about the type, such as maximum and minimum values or size of the type can be retrieved in Lua code. For example, on this machine, `print(int.bytes)` in the terra toplevel outputs 4. This allows the generation of generic data types, similar to using C++ templates.

4.3 Specialization and Typechecking

The interaction between Terra and Lua has three important phases: Specialization, Typechecking

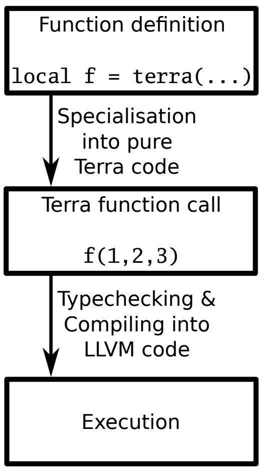

Figure 2: The compilation process of Terra

and, finally, compiling.
When Terra code is first defined, specialization is executed. This means that all Lua variables and splices in the code are replaced with their values (i.e. Lua code is executed to get its value). This process may fail, for example when a variable that is not bound is encountered in the Terra code. After specialization, the function consists only of Terra code.
It is important that this specialization step occurs when the Terra function is defined: If specialization occurred when the Terra function is called, lexical scoping for the Lua values would not be preserved, and code could refer to Lua values that are not defined at the point of definition. When a Terra function is run, it is typechecked and compiled into LLVM code, and then run by LLVM. For an overview of the process, see figure 2.
Using the methods `printpretty()` and `disas()`, both the generated Terra code after specialization and the generated assembler code after compilation can be examined, as is shown in listing 6. Also, using the `saveobj()` method, the compiled Terra function can be written to an object file and reused by other C code. Executable files can be written as well, allowing Terra to generate exceutables that do not contain any trace of the Terra runtime any more.

```
local a = 1
local b = "hanspeter"
cio = terralib.includec("stdio.h")

local terrafn = terra()
  cio.printf("A␣is␣%d␣and␣b␣is␣%s.\n", a, b)
end

terrafn:printpretty()
terrafn:disas()
-- Generates an executable file that calls terrafn
terralib.saveobj("out", {main = terrafn})
```

Listing 6: Examining generated code

5 Applications

Multi-stage programming is of great interest when performant programs are required. Applications for multi-staging techniques exist in multiple fields. A first simple example is the generation of staged power functions in listing 4. When a specific power function (e.g. `pow10()` is needed multiple times, a pre-generated function can be more efficient than a compiled function (provided the power function is not called with a constant first argument). Because the programmer has more information about the call contexts of the function, he can generate code that is more efficient than regular compiled code.
Having control over code generation allows the programmer better control over performance optimizations. While often, a smart compiler can generate performant code for a variety of systems, the programmer often has a better understanding of runtime behavior of code. This allows performance optimizations that are not immediately obvious to compilers. Using code generation, optimizations can be stated on a higher level, without unduly complicating the program.
An example presented by DeVito et al[6] is loop blocking: changing the iteration order of a computation-intensive loop can produce more performant cache behavior of the program. A sophisticated compiler would most likely execute this optimization as well, but by using multi-stage programming, the optimization remains under control of the programmer. This allows application of auto-tuning techniques, such as evaluating different blocking schemes in one program. By auto-tuning a matrix multiply operation, a high-performance matrix multiply operation was generated by DeVito et al. with comparable performance to commercial linear algebra packages. The program was

only 200 lines of code in all, much smaller than the other evaluated packages.

Other optimizations that can benefit from these auto-tuning techniques are register blocking (optimize the code so variables all fit inside the registers of the target architecture), vectorization (Terra includes support for vector operations), loop unrolling or instruction reordering. While applying these transformations requires a good understanding of optimization techniques in general, multi-stage techniques allow implementation of these techniques in a general way. Optimization methods could even be exposed in a library, which would make implementation of auto-tuning software even easier.

Apart from low-level optimizations such as those described above, multi-stage programming is also useful for writing compilers or implementing domain specific languages in an efficient manner. By emitting code instead of evaluating code, a compiler for a language can be written in a simple manner, simply by inserting staging constructs in the correct places.

```
--AST format: {op = <cmdname>,
--arg1 = <first argument>, arg2 = <second argument>
function eval(command, env)
  if command.op == "True" then return(true)
  elseif command.op == "False" then return(false)
  elseif command.op == "And" then
    return (eval(command.arg1, env) and
          eval(command.arg2, env))
  -- Some code omitted
  elseif command.op == "Forall" then
    local truevalue =
      eval(command.arg2, ext(env, command.arg1, true))
    local falsevalue =
      eval(command.arg2, ext(env, command.arg1, false))
    return truevalue and falsevalue
  elseif command.op == "Var" then
    return(env(command.arg1))
```

Listing 7: Non-staged QBE interpreter

```
function generate_code(code, stack, vars,
                   varsize, frame)
  if code.op == "True" then
    return quote
      stack:push(true)
    end
   elseif code.op == "False" then
    return quote
      stack:push(false)
    end
  elseif code.op == "And" then
    return quote
      [ generate_code(code.arg1, stack,
        vars, varsize, frame) ]
      [ generate_code(code.arg2, stack,
        vars, varsize, frame) ]
      stack:push(stack:pop() and stack:pop())
    end
-- Code omitted
  elseif code.op == "Var" then
    local i2 = frame[code.arg1][1]
    return quote
     stack:push(vars[ [i2] ] )
    end
  end
end

function compile(code, vars)
  local frame = {}
  for k,v in pairs(vars) do
    frame = extend_frame(v, frame)
  end
  return terra(vars: &bool, varsize: int)
             : {bool}
    var s: &Stack = newstack()
    [ generate_code(code, s, vars,
              varsize, frame) ]
    return s:pop()
  end
end
```

Listing 8: Staged QBE compiler for a stack machine

As an example, consider a program for evaluating quantified boolean expressions (expressions such as $\forall x : \neg x \vee x$). An interpreter would look something like the code in listing 7. The interpreter is very easy to understand and easy to verify. However, in general interpreters are not very efficient. Using staged programming techniques, a compiler for the same program can be written. It is very similar to the original simple interpreter, but performs better. For some example code showing the staged compiler, see listing 8. While variable handling has become a bit more difficult, the general structure of the compiler is still recognizably similar to the original interpreter. While compilers can theoretically be generated automatically from partially evaluated interpreters (see [9]), the multi-stage approach is easier to understand for the programmer, and creates a useful starting point for improving the compiler. DSLs can be useful for many different causes, and performance is often a concern. A current example is the implementation of nftables[4], a network filtering software, where filter expressions are compiled into efficient byte code.

6 Related Work

Terra is a multi-stage language geared towards creating high-performance programs by parametrizing the execution of a low-level language via a higher-level language. While Terra and Lua are integrated relatively closely, they are nonetheless different languages. Terra is a *heterogeneous* multi-stage language.

Heterogeneous multi-staged environments have been used before in order to generate high-performance code. This staging approach is used with much success in the Fast fourier transform library FFTW[8], the linear algebra library ATLAS[28] or SPIRAL[19], an autotuning signal-processing library. The multi-staging approach used in these libraries is comparable to SQL query optimization: A plan is generated using some building blocks suitable for the solution weighted by a cost function, and then it is executed. The generation and execution of the plan are the stages in this process. This approach aims to generate optimized software without requiring intimate knowledge of the target.

However, the mentioned packages only solve very specific problems, and the generated plan is not accessible to the user (or at least not meant to be accessed). Also, writing plans directly is not supported. Because of this, these packages cannot be considered implementations of heterogeneous multi-stage programming languages (they use features of multi-stage programming, however!).

In contrast to heterogeneous multi-stage languages, several homogeneous multi-stage languages exist: MetaML[25], MetaOCaml[24], MetaHaskell[15] and Metaphor[17] as well as several others[18][20][13]. In these languages, the meta-language and the object language are the same. Also, all of these languages use static typing. These choices have several advantages: Using the same language for both meta- and object program allows an arbitrary number of stages. In a heterogeneous language, this is only possible if meta- and object languages able to embed each other. Also, with static typing it is possible to ensure that generated code is also type-correct, something that Terra can not guarantee: Since Lua is dynamically typed, the generation of code can potentially fail (namely, during specialization and typechecking). Using static types, it is possible to ensure certain guarantees for generated code. For a closer look at problems arising in statically typing multi-stage code, refer to [23].

Some multi-staging approaches in mainstream languages also exist. The C++ template system deserves special mention: Using the template system, arbitrary code can be executed at compile time. It is, for example, possible to generate a list of prime numbers. The C++ template system is, in fact, a turing-complete functional language executed at compile time. For a closer description of C++ template metaprogramming, see [26]. Template metaprogramming has also been introduced into other languages. TemplateHaskell[22] for example formalizes a language for preprocessing the source code of Haskell programs. TemplateHaskell is integrated in current versions of the Glasgow Haskell Compiler. OCaml also supports a preprocessing system called Camlp4[5]. Note that these template systems do not offer the full flexibility of a multi-staging language: Code is generated at compile time and executed at runtime, while in multi-stage languages, code can be executed arbitrarily. Programs such as auto-tuners that interleave code generation and execution can not be realized in template systems.

Macros are a technique that is similar to the template metaprogramming approach. In languages with “proper” macros (unlike the C preprocessor), macros are written in the language being compiled, and are expanded at compile time, taking the unevaluated source code as an argument. This allows generation of programs at compile time, similar to multi-staged programming. In fact, [10] argues that macros are a subset of multi-staged computation.

7 Conclusion

The heterogeneous multi-staging approach of Terra seems a promising approach to generating high-performance code, while still allowing the code to be well abstracted. Using the multi-staging features of Lua, high performance programs can be created by generating code that is optimal for the current runtime environment. Examples of such optimisations are loop blocking (dependent on the size of caches) or instruction vectorisation (dependent on the instruction set the processor supports). The Terra language makes it easy to perform these and similar low-level optimizations from a high-level standpoint, without losing generality. In contrast to other performance-oriented code generation tools, Terra allows these optimizations without using supplemental tools.

On the other hand, using a heterogeneous multi-stage language entails translation between those two languages. While the conversion between Terra and Lua values is handled transparently for simple types (like strings and numbers), conversion has to be done manually in many cases. For more complex data structures, developers have to implement their own conversion functions. Here, it would be interesting to implement better conversion routines between Lua and Terra, and reducing the need for explicit type specification when transferring data between Lua and Terra.

The decision to use dynamic typing for the upper-stage language was most probably motivated by challenges in static typing for multi-stage languages (compare [3]). While dynamic typing for the upper stage language eases implementation without concern for edge cases in the type system, it introduces potential for error. Statically typed multi-stage languages like MetaOCaml guarantee that only type-correct code can be generated, but have to deal with some unique complications (such as scope extrusion). A statically typed multi-stage language that shares Terra's C compatibility and staging approach would ease interaction between the two stages, and allow the same optimizations that Terra does. However, most of the use of multi-stage languages has been academic. Also, many languages are no longer supported and developed by their authors.

Since most research has occurred in the realm of homogeneous multi-stage languages, the release of Terra will hopefully give a boost to research in the area of heterogeneous multi-stage languages. With infrastructure such as LLVM, we will hopefully see other multi-stage languages that allow performance optimizations similar to Terra in a statically typed setting.

References

[1] Ned Batchelder. Cog: a code generation tool. `http://nedbatchelder.com/code/cog/`.

[2] Alan Bawden et al. Quasiquotation in Lisp. In *PEPM*, pages 4–12. Citeseer, 1999.

[3] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In *Automata, Languages and Programming*, pages 25–36. Springer, 2000.

[4] Jonathan Corbet. The return of nftables. `http://lwn.net/Articles/564095/`, August 2013. Accessed: 15.1.2014.

[5] Daniel de Rauglaudre. Camlp4. `http://pauillac.inria.fr/camlp4/`, Jan 2014. Accessed: 14.1.2014.

[6] 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.

[7] Free Software Foundation. Autogen: the automated text and program generation tool. `https://www.gnu.org/software/autogen/`. Accessed: 27.11.2013.

[8] Matteo Frigo and Steven G Johnson. FFTW: An adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[9] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[10] Steven E Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in macroml. In *ACM SIGPLAN Notices*, volume 36, pages 74–85. ACM, 2001.

[11] Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo. The Lua programming language. `http://www.lua.org/`, 1993. Retrieved: 1.12.2013.

[12] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN*

Symposium on Principles of Programming Languages, POPL '86, pages 86–96, New York, NY, USA, 1986. ACM.

[13] Jurgen Kleinoder and Michael Golm. MetaJava: an efficient run-time meta architecture for Java$^{\text{TM}}$. In *Object-Orientation in Operating Systems, 1996., Proceedings of the Fifth International Workshop on*, pages 54–61. IEEE, 1996.

[14] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[15] Geoffrey Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*, pages 311–322, Copenhagen, Denmark, 2012.

[16] Randy Meyers. The new c: X macros. `http://www.drdobbs.com/the-new-c-x-macros/184401387`, May 2001. Accessed: 11.1.2014.

[17] Gregory Neverov and Paul Roe. Metaphor: a multi-stage, object-oriented programming language. In *Generative Programming and Component Engineering*, pages 168–185. Springer, 2004.

[18] Massimiliano Poletto, Wilson C Hsieh, Dawson R Engler, and M Frans Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):324–369, 1999.

[19] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[20] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.

[21] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer Berlin Heidelberg, 2001.

[22] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[23] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: axiomatization and type safety. In KimG. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929. Springer Berlin Heidelberg, 1998.

[24] Walid Taha, C Calcagno, L Huang, and X Leroy. MetaOCaml: a compiled, type-safe multi-stage programming language. 2001.

[25] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1):211–242, 2000.

[26] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[27] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[28] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3 – 35, 2001. New Trends in High Performance Computing.