First-class Runtime Generation of High-performance Types using Exotypes

Zachary DeVito Daniel Ritchie Matt Fisher Alex Aiken Pat Hanrahan
Stanford University
(zdevito|dritchie|mdfisher|aiken|hanrahan)@cs.stanford.edu

Abstract

We introduce *exotypes*, user-defined types that combine the flexibility of meta-object protocols in dynamically-typed languages with the performance control of low-level languages. Like objects in dynamic languages, exotypes are defined programmatically at runtime, allowing behavior based on external data such as a database schema. To achieve high performance, we use staged programming to define the behavior of an exotype during a runtime compilation step and implement exotypes in Terra, a low-level staged programming language.

We show how exotype constructors compose, and use exotypes to implement high-performance libraries for serialization, dynamic assembly, automatic differentiation, and probabilistic programming. Each exotype achieves expressiveness similar to libraries written in dynamically-typed languages but implements optimizations that exceed the performance of existing libraries written in low-level statically-typed languages. Though each implementation is significantly shorter, our serialization library is 11 times faster than Kryo, and our dynamic assembler is 3–20 times faster than Google's Chrome assembler.

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

General Terms Design, Performance

Keywords Lua, Staged computation, DSL

1. Introduction

A language's object representation and implementation has a significant effect on both the concepts that can be easily expressed and the ease of generating high-performance code. For instance, Norvig [21] found that, of the 23 original design patterns proposed by Gamma et al. [8], 16 became simpler or are implementable as libraries using the built-in language features of Lisp or Dylan. One reason is that many dynamic languages such as Lisp, Python, or Lua support so-called *meta-object protocols*, meaning there is a mechanism for the user to programmatically modify the semantics and implementation of user-defined types [1, 14, 15]. Higher-level policies such as inheritance or accessor permissions can be defined on top of these mechanisms, giving the programmer great flexibility in defining object behavior.

However, when an object's behavior and in-memory representation are defined dynamically, it is difficult to perform some optimizations, resulting in performance losses. For instance, in Section 4, we implement a microbenchmark of an Array object that forwards methods to each of its elements. The JIT-compiled Lua version runs 18 times slower than equivalent C++ code due to object boxing and dynamic dispatch. JIT compilers can optimize some dynamic patterns [6, 13], but it is difficult to know if a pattern will result in high-performance code.

In this work, we introduce *exotypes* as a way to achieve the expressiveness of dynamic languages while retaining the performance of statically-typed objects. Exotypes combine meta-object protocols with multi-stage programming to give the programmer more control over the code's performance. Rather than define an object's behavior as a function that is evaluated at runtime, an exotype describes the behavior with a function that is evaluated once during a staged compilation step. These functions *generate* the code that implements the behavior of the object in the next stage rather than implementing the behavior directly. This design allows the programmer to optimize behavior and memory layout before any instances of the object are used.

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

We implement this approach by extending the object system of the Terra language. Terra is a staged, low-level system programming language similar to C that is embedded in Lua, a high-level dynamically typed language [7]. Terra's user-defined types are replaced with exotypes that are defined *external* to the Terra language using a meta-object protocol based in Lua. Types are defined via user-provided property functions that describe their behavior and in-memory layout using multi-stage programming. Terra has a low level of abstraction, so it is possible to control the performance of the staged code. It is also a staged language, so new exotypes can be defined dynamically over the course of the program. Higher-level features, such as object serialization or polymorphic class systems,

PLDI'14, June 11–18 2014, Edinburgh, United Kingdom..

http://dx.doi.org/10.1145/2594291.2594307

can be built on top of these types. We present the following contributions related to exotypes:

- We introduce the concept of an exotype and present a concrete implementation in the Terra compiler based on programmatically-defined properties queried during typechecking.
- We show that high-level type features such as type constructors can be created with exotypes. When specified in a well-behaved manner, independently-defined type constructors can be composed.
- We evaluate the use of exotypes in several performance-critical scenarios: serialization, dynamic assembly, automatic differentiation, and probabilistic programming.

In the scenarios we evaluate, we show how we can achieve expressiveness similar to libraries written in dynamically-typed languages while matching the performance of existing implementations written in statically-typed languages. The added expressiveness makes it feasible to implement aggressive optimizations that were not attempted in existing static languages. Our serialization library is 11 times faster than Kryo (a fast Java serialization library). Our dynamic x86 assembler can assemble templates of assembly 3–20 times faster than the assembler in Google Chrome, and our implementation of a probabilistic programming language runs 5 times faster than existing implementations.

2. Background

Meta-object protocols. Modern dynamic languages allow programmatic definition of object behavior. For instance, Python provides *metaclasses* which can override the default behaviors of method definition and invocation, and CLOS allows for the dynamic specification of all behavior of objects using so-called meta-object protocols [1, 15]. The Lua language uses a meta-object protocol based on *metatables* to extend the normal semantics of objects [14]. Metatables are Lua tables containing functions that define new semantics for default behaviors. For instance, we can change the behavior of the table indexing operator `obj.field` by setting the `__index` field in a metatable:

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

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

We use a meta-object protocol defined using Lua tables to describe the behavior of exotypes. However, the behaviors in exotypes are expressed using staged programming and queried before code that uses the objects is compiled. While most meta-object protocols are applied dynamically, some, such as those in Open-C++, are applied statically during compilation [3]. In these systems, no new types are defined at runtime. Exotypes blend the two approaches. New types can be created and compiled as the program runs, but since exotype behavior is described with staged programming of a low-level language (Terra), the programmer retains control over low-level representation and implementation.

Multi-stage programming. Our implementation of exotypes relies on multi-stage programming (MSP) to dynamically generate expressions that implement object behavior. MSP as described by Taha and Sheard allows the programmer to separate a program into multiple phases using explicit program annotations [29]. This design can be viewed as an abstraction over code generation and compilation. An earlier stage of the program can generate and compile code that runs at a later stage. By explicitly representing these compilation steps, MSP gives the programmer precise control over generated code and allows code generation to be based on dynamic information.

Our implementation builds on the Terra language, a staged programming language embedded in Lua and designed for generating high-performance code [7]. Lua is used for constructing Terra programs and writing high-level program transformations. Terra is a low-level language with semantics and types similar to C. Since it has a low level of abstraction, it is relatively easy to reason about and tune the performance of generated Terra code.

A Terra function is defined in Lua code using the `terra` keyword (in Lua, a function is normally created using `function`). A Terra function can be called directly from Lua:

```
terra powf(v : double, N : int)
    var r = 1.0
    for i = 0,N do r = r * v end
    return r
end
powf(2,3) --terra function called from Lua
```

Staged programming in Lua and Terra involves two phases of meta-programming. First, untyped Terra expressions are constructed using *quotations* and stitched together using *escapes* in a process we call *specialization*. Second, type-level computation can be carried out during typechecking with user-defined *type-macros*, which we will use to implement exotypes. The interaction between these phases is summarized in Figure 1 (left).

A quotation (the backtick operator `` `exp ``, or the block structured `quote <exps> end`) used in Lua code creates an unevaluated Terra expression, and an escape (the bracket operator `[lua_exp]`) used in Terra code evaluates `lua_exp` and splices its result (normally a Terra quotation) into the surrounding Terra code. Consider how to use these operators to generate a specialized version of `powf` for a particular value of `N`:

```
function genpowf(N)
    local function genexp(vr)
        local r = `1.0
        for i = 1,N do r = `([r] * [vr]) end
        return r
    end
    local terra powfN(v : double)
        return [genexp(`v)]
    end
    return powfN
end
pow2 = genpowf(2)
print(pow2(3)) -- '9'
```

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

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

To support our implementation of exotypes, we use an additional operator, the *type macro* that allows for user-defined behav-

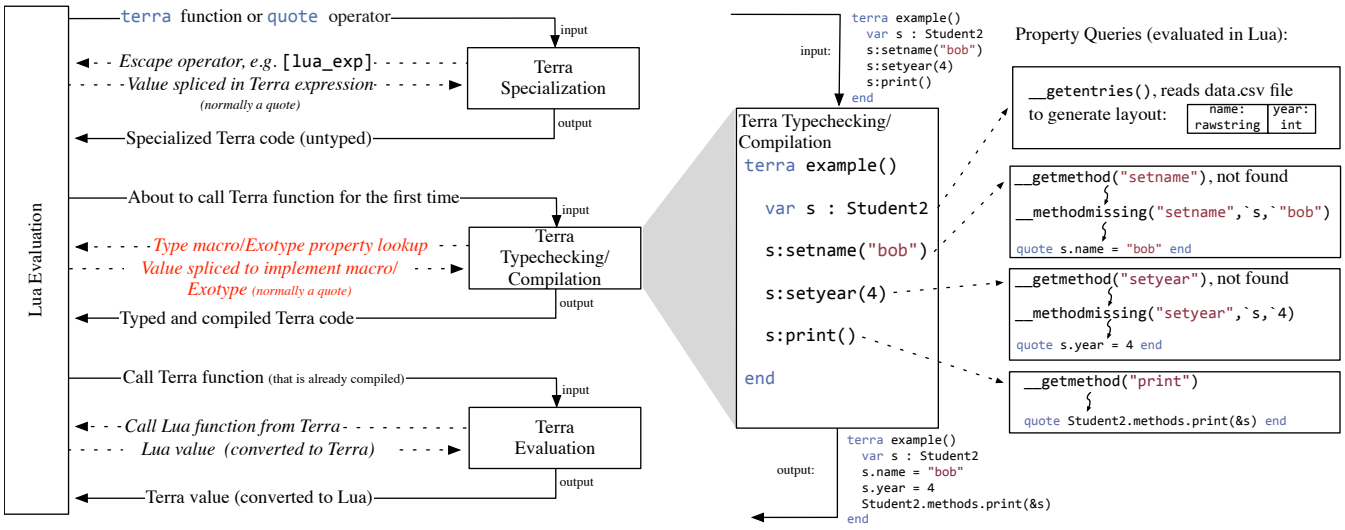

Figure 1. Phases of evaluation of a Lua-Terra program with exotypes (left), and an example of the typechecking process for the `Student2` exotype (right). New interactions used to implement exotypes are highlighted in red.

ior during typechecking. These macros can be used in Terra code like a function, but they are evaluated when the Terra code is typechecked. Unlike escapes and quotes, which are used to create and stitch code together before typechecking, type macros have access to their arguments' types, and are used to generate behavior based on the types as shown in this example:

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

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

3. Exotypes Interface

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 1. A selection of exotype properties and example operations that cause the type checker to invoke them.

type, in which we use the `__methodmissing` property to create setter methods (e.g., `setname`) for each field.

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

The typechecking process for example code that uses `Student2` is illustrated in Figure 1 (right). When the typechecker sees an instantiation of the type, it queries `__getentries` to get its memory layout. For `Student2` this will load the `data.csv` file to determine the layout. If the typechecker sees a call to method like `setname` which is not in `Student2.methods`, then the `__getmethod` property will query the `__methodmissing` property for its behavior. Since `__methodmissing` is defined as a type macro, it will be evaluated during the typechecking process to produce a Terra quotation that implements the behavior.

Other properties define the behaviors for built-in operators (e.g. `__add` for `+`), and how to convert between user-defined types (`__cast`). Table 1 lists some common properties and the operations they define.

Defining behavior programmatically allows generic behaviors to be expressed concisely. For instance, we can interface with externally-defined class systems. Objective-C is an extension to C that adds objects similar in behavior to Smalltalk. We can embed Objective-C objects by creating an exotype `ObjC` wrapper:

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

When the typechecker sees a method called on an `ObjC` object (e.g. `obj:init(1)`), its `__methodmissing` property will insert code to call the Objective-C runtime API, e.g:

```
C.objc_msgSend(obj.handle,init_name,1)
```

Staging of properties gives the programmer control over the performance of the type. Since behavior can be defined in type macros that are evaluated during typechecking, information known statically can be pre-computed. In the `ObjC` type, the method name `sel` is known during typechecking, so we can pre-compute the Objective-C method selector (line 5), which makes the expression executed at runtime (line 7) faster.

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

4. Example: Array(T)

Exotypes allow us to express the generic behavior of objects in a way similar to meta-object protocols in dynamic languages while still achieving the the performance of low-level languages. Consider how to represent the type constructor `Array(·)`, which takes a type `T` and produces a new type, `Array(T)`, that holds a collection of `T`s. In addition, for each method `m` of `T`, `Array(T)` has its own method `m` that invokes the original `m` on each member of the array. This is a *Proxy* design pattern [8], where methods on one object are forwarded to methods on another. This specification is different from that in most functional languages, where this behavior is implemented as a higher-order `map` function, but is widely used in array-based languages such as APL and R to concisely operate on collections.

Proxy patterns are difficult to express generically in many modern languages. Despite the fact that proxies are a common pattern in object-oriented computing, programmers using C++ or Java must duplicate the method names of the object in the proxy. Some languages have added more advanced features to support proxies and other design patterns. For instance, Hannemann and Kiczales show that AspectJ can automate some aspects of proxies, but not in a way that is generically reusable [12]. Expressing proxies in Scala requires the use of advanced features such as its macro facilities and `Dynamic` type trait that were only added in version `2.10` of the language [2].

In contrast, proxies are relatively easy to implement in dynamic languages. As an example, `Array(·)` can be implemented concisely using metatables in Lua:

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

When a missing method on an array is referenced, the metatable's `__index` field (line 12) causes `createmethod` to be called, which generates an implementation for the method (line 2) that loops over the objects in the array forwarding the method call to each.

While simple, it is hard to control the performance of the object. Methods are looked up dynamically on each object in the array. The objects themselves are boxed, limiting memory locality. We implemented a micro-benchmark that invoked a simple counter function on each member of an array object and evaluated it using LuaJIT, a state-of-the-art tracing JIT [23]. Despite tracing and compiling the loop, it still performs 18 times slower than hand-written C++ that implements the proxy by hand due to the overhead of guards and unboxing of objects.

With exotypes, we can specify `Array(·)` almost as concisely as the Lua code, but staging of its behavior and layout allows us to remove the inefficiencies:

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

The `__methodmissing` type macro performs a similar purpose to the `createmethod` function in the Lua example. It generates the code that loops over elements of the array and forwards the method call, but it does so during typechecking rather than evaluation. Furthermore, the declaration of `ArrayImpl` provides a concrete layout for the type before it is compiled.

Since the layout of `ArrayImpl` is described before compilation, we can store the array's `T` objects unboxed rather than as an array of pointers. Furthermore, the forwarded calls to `methodname` on line 9 are resolved at compile time and inlined. In our micro-benchmark, this staging allowed us to generically generate the same code as the hand-written C++ proxy, and run at the same speed.

5. Composability

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

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

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

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

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

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

In the original publication on Terra [7] types were described using eagerly built tables that specified the layout and methods, in addition to ad-hoc user-defined callbacks invoked by the typechecker. Our experience with issues causing subtle cycles while building type constructors such as `Array(·)` led us to replace this design with exotypes based on lazily queried properties.

Lazily queried properties also make it possible to create types that have an unbounded number of behaviors. For instance, the Objective-C wrapper object presented previously can respond to an unlimited number of messages. Despite being unbounded, these methods can compose with other type constructors. For instance, it is possible to make an `Array(ObjC)` that stores OS windows and call `windows:makeKeyAndOrderFront(nil)` to focus them. In this case, the method `makeKeyAndOrderFront` is requested by the typechecker via the `__getmethod` property of `Array(ObjC)`, which will in turn query the `__getmethod` property of `ObjC` to generate the method call. If `Array` required all methods to be available up-front, these two types would not compose.

6. Formalization of Properties

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

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

Terra is evaluated in multiple phases. To keep our formalization of exotypes simple, we focus on the typechecking and evaluation of Terra code with exotype property queries. In particular, we omit the machinery for quotes, escapes, and specialization used to generate Terra code, which can be found in previous work [7], and assume that an environment $\mathtt{P}$ that holds a mapping between exotypes and their properties is already constructed. This environment would normally be created in Lua by API calls that create types and set entries in their metamethods table.

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

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

Types are either a base type `Int`, a function $\mathtt{T} \to \mathtt{T}$, or an exotype represented by a type identifier `t`. An exotype constructor ($\mathtt{c}_{\mathtt{t}}\ \dot{\mathtt{e}}$) takes an expression $\dot{\mathtt{e}}$ to initialize the internal value of the exotype, and $\mathtt{unwrap}\ \dot{\mathtt{e}}$ retrieves the internal value. The environment $\mathtt{P}$ holds a mapping from an exotype `t` to its *property lookup function*, $\mathtt{P}(\mathtt{t}) = \lambda \mathtt{x}.\mathtt{e}$. In real Terra, each property is a different function, but

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

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

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

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

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

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

Figure 2. Typing rules, $P\ \Gamma \vdash \dot{e} : T$, for ExoTerra expressions. $\vdash'$ refers to the same rules but with TyExoApp removed.

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

$$\frac{P\ \dot{e} \longrightarrow_T P\ \dot{e}'}{P\ \dot{C}[\dot{e}] \longrightarrow_T P\ \dot{C}[\dot{e}']}\ (\textsc{TCtx}) \qquad \frac{P\ (\lambda \dot{x} : T.\dot{e})\ \dot{v} \longrightarrow_T P\ \dot{e}[\dot{v}/\dot{x}]}{P\ \mathsf{unwrap}\ c_t\ \dot{v} \longrightarrow_T P\ \dot{v}}\ (\textsc{TApp})\ (\textsc{TUnwrap})$$

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

Figure 3. Rules, $P\ \dot{e} \longrightarrow_T P\ \dot{e}'$, for evaluating ExoTerra expressions.

$$C \quad ::= \bullet \mid C\ e \mid v\ C \mid \mathsf{prop}\ v_0 ... v_{i-1}\ C\ e_{i+1} ... e_n \mid \mathsf{runprop}\ S\ C$$
$$C' \quad ::= \bullet \mid C'\ e \mid v\ C' \mid \mathsf{prop}\ v_0 ... v_{i-1}\ C'\ e_{i+1} ... e_n$$

$$\frac{P\ e \longrightarrow_L P\ e'}{P\ C[e] \longrightarrow_L P\ C[e']}\ (\textsc{LCtx}) \qquad \frac{C \neq \bullet}{P\ C[\mathsf{error}] \longrightarrow_L P\ \mathsf{error}}\ (\textsc{LError})$$

$$P\ (\lambda x.e)\ v \longrightarrow_L P\ e[v/x]\ (\textsc{LApp}) \qquad P\ \mathsf{runprop}\ S\ v \longrightarrow_L P\ v\ (\textsc{LRunProp})$$

$$\frac{(v_0, ..., v_n) \in S}{P\ \mathsf{runprop}\ S\ C'[\mathsf{prop}\ v_0 ... v_n] \longrightarrow_L P\ \mathsf{error}}\ (\textsc{LPropCycle})$$

$$\frac{(t, v_1, ..., v_n) \notin S \qquad P(t) = \lambda x.e}{P\ \mathsf{runprop}\ S\ C'[\mathsf{prop}\ t\ v_1 ... v_n] \longrightarrow_L P\ \mathsf{runprop}\ S\ C'[\mathsf{runprop}\ (S \cup \{(t, v_1, ..., v_n)\})\ ((\lambda x.e)\ v_1 ... v_n)]}\ (\textsc{LProp})$$

Figure 4. The rules, $P\ e \longrightarrow_L P\ e'$, for evaluating ExoLua.

this is only for convenience. In our formalism, the kind of property is passed as an argument. The property lookup functions are written in ExoLua, based on the untyped lambda calculus:

$$v \quad ::= \quad T \mid \lambda x.e \mid \dot{e}$$
$$e \quad ::= \quad v \mid e\ e \mid x \mid \mathsf{error} \mid \mathsf{prop}\ e_0 ... e_n \mid \mathsf{runprop}\ S\ e$$

ExoTerra types (T) and expressions ($\dot{e}$) are values in ExoLua so that they can be returned from property queries. An exception expression, `error`, models the errors that occur when we find a cyclic property query. The `prop` form is used to query a property of a type. Its first argument e_0 should evaluate to an exotype identifier `t`, and $e_1 ... e_n$ are additional arguments describing the query. Properties are evaluated in the context of querying other properties. This evaluation is formalized with the `runprop S e` expression, which evaluates a property lookup expression `e` in the context `S`, where `S` is a set of properties. Each property in `S` is already being queried when `e` is requested, and has the form $(v_0, ..., v_n)$.

The rules for typechecking ExoTerra are shown in Figure 2. Rule TyCtor illustrates how the typechecking process queries the property function to retrieve information about the type. In particular, TyCtor asks the property function for the type of the concrete value that will represent the exotype, and makes sure it matches the type of the expression used to initialize it. Rule TyExoApp shows how a property function defines the behavior of an exotype when it is applied like a function. It queries the exotype for its behavior when applied to a value of type T' (`runprop` $\emptyset$ `prop` λ`x.e apply` T'). This query should produce an implementation of behavior in the form of a function ($\dot{e}''$). This function takes the exotype $\dot{e}$ and the argument $\dot{e}'$ to compute the value of the expression. We check $\dot{e}''$ with modified typing rules $\vdash'$ which omit TyExoApp. This change prevents the implementation $\dot{e}''$ of an exotype from relying on exotype behavior itself, which can prevent typechecking from terminating if, for instance, a query about exotype application included the same application in its implementation. This behavior does not restrict what implementations can be expressed, since the property generating the implementation function can query exotypes for the appropriate behavior. Rules for evaluating ExoTerra are presented in Figure 3 and show how the results of property queries will be applied to evaluate Terra code.

The rules for evaluating ExoLua are shown in Figure 4. Exceptions abort the computation (LError). The rules LProp and LPropCycle perform property queries. A property is computed as the nested application $v_0 ... v_n$, where v_0 is the property lookup function for type `t`. We say that the tuple $(t, v_1, ..., v_n)$ is the property being queried. For example, the expression `P(t) apply T` is a query of the (t, apply, T) property. Property statements (`prop`) can only step inside of a `runprop` rule which specifies the set `S` of active property lookups. If the same property is already being queried, it is in set `S` and will evaluate to `error` (rule LPropCycle), otherwise the property will be evaluated in a new `runprop` context that records the fact that the property is currently being queried (rule LProp). The values in property queries can be functions; for the purposes of `S`, we consider two lambda terms equal if they are equivalent up to alpha conversion. Given this formalization, we can define sufficient conditions to ensure that a property lookup during typechecking will terminate:

- *Individual termination.* A property evaluation `e` in a program $P\ C[\mathsf{runprop}\ S\ e]$ reduces to $P\ C[v]$ or $P\ \mathsf{error}$ assuming that all of the subsequent property evaluations that it evaluates ($P\ C[\mathsf{runprop}\ S\ C'[\mathsf{prop}\ v'_0, ...v'_n]]$) also reduce to values ($P\ C[\mathsf{runprop}\ S\ C'[v']]$) or $P\ \mathsf{error}$.
- *Closed universe.* There exist a finite number N of unique properties of the form $(v_0, ..., v_m)$ that can be queried.

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

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

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

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

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

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

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

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

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

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

7. Examples and Evaluation

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

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

7.1 Serialization

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

One solution is to provide robust libraries that attempt to address every possibility. Google's Protocol Buffers provide cross-language data-description and versioning [11]. Java includes a built-in library for serialization that can serialize entire object graphs and ensure type safety [22]. To optimize performance, some libraries such as the Kryo library for Java generate specialized serialization code for each type ahead-of-time [26]. The robust features in these libraries often make them difficult to customize, which is unfortunate, since advanced features such as supporting object graphs can incur runtime overhead.

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

We created a generic exotype suited to serialize this kind of data. It responds to two polymorphic methods `write` and `read`. A partial list of the code for the implementation of `write` is shown in Figure 5, focusing on serialization of structs (other exotypes). Code is generated for each type seen in the object tree. For `struct` objects, it queries the layout of the `struct` to generate code for each of the entries.

Performance for a 1.28MB scene is shown in Figure 6. We compare against Java's native serialization, the state-of-the-art Kryo library, and Google's C++ implementation of protocol buffers. Objects were serialized to a pre-allocated buffer in memory so that the benchmark would capture encoding time rather than buffer allocation/resizing time. For Java implementations, the JIT was

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

Figure 5. Implementation of a generic serializer, focusing on code for serializing aggregate types.

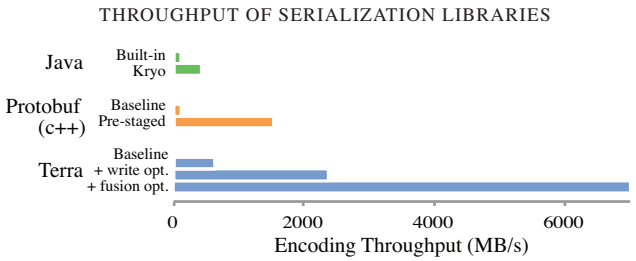

Figure 6. Throughput of scene graph serialization. Our specialized exotype implementation performs an order of magnitude faster.

warmed up by serializing the entire scene 250 times before timing. For Kryo, classes were pre-registered, support for object graphs was disabled, and `UnsafeMemoryOutput` was used to increase performance.

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

Given a specific use-case, we can apply more aggressive optimizations. Our baseline implementation calls a user-provided function pointer to write data. We can turn our serialization type into a type-constructor that takes the write function as an argument. This change (write opt.) allows inlining calls to the write function, increasing performance to 2.37 GB/s. Our baseline implementation also writes each element of a struct individually, allowing for customized writers for specific objects. However, in this example, no custom writers are needed, so the library can apply aggressive fusion to the writes. Rather than copy each struct element individually, it copies the entire struct at once. Any objects pointed to by elements of the struct are then written afterward. Using vectorization, these larger copies are efficient, increasing the performance to 7.00GB/s (fusion opt.) or 11 times faster than Kryo. As an upper bound on performance, simply copying 1.28MB runs at 15GB/s.

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

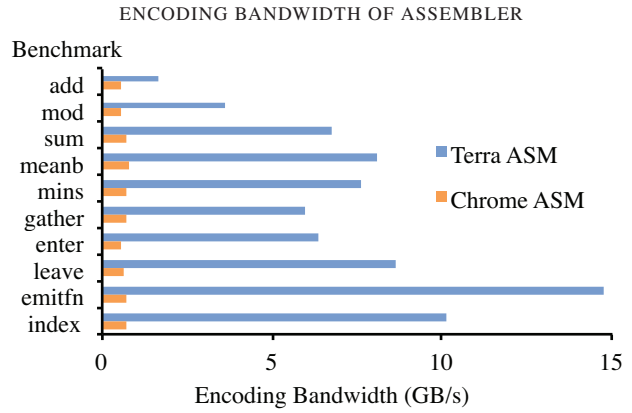

Figure 7. Throughput of assembler. Benchmarks are ordered by increasing template size. Larger template sizes increase the effect of the Terra assembler's fusion optimization.

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

7.2 Dynamic x86 Assembly

Dynamic assemblers form the basis of JIT compilers. Unlike normal assemblers which run during compilation, dynamic assemblers are used at runtime to dynamically generate code. A JIT compiler uses the assembler to translate its own intermediates into executable machine code. Higher-level JIT-compiled languages may emit *templates* of x86 assembly that correspond to a single high-level instruction. For instance the Riposte JIT [30] for vector code in R uses the following template to emit a vector gather:

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

The gather takes 2 addresses in `RegA` and then loads the two values in `RegR`, requiring 5 instructions total.

The performance of a JIT depends on compilation speed, so it is important for the `Assembler` type to be efficient. Riposte uses the assembler used in Google Chrome [10] to get high performance. To ensure speed, each instruction is explicitly implemented as a method on the assembler object. As an example, here is the implementation of the `movlpd` instruction:

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

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

Another approach is to describe the instruction concisely in a small language, similar to how string matching can be encoded with regular expressions. LuaJIT's DynAsm library [23] takes this approach to describe the `movlpd` instruction in a table:

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

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

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

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

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

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

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

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

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

This function fuses the assembly into a single template copy followed by patch-up instructions to insert `RegR`, `RegA`, and `disp`. The resulting code runs at 5.96 GB/s, or 8.3 times faster than the hand-written Chrome version.

To validate the approach, we rewrote 10 kernels taken from the Riposte JIT compiler using our dynamic assembler written in Terra and compare their performance to those of the original code which uses the Chrome assembler. Figure 7 shows the results for each kernel. While the Chrome assembler always emits code at a rate of 700MB/s, the Terra assembler can perform anywhere from 1.64 GB/s to 15 GB/s, depending on the amount of instruction fusion that is possible. For instance, the `add` instruction only includes two x86 instructions, and one of them is only emitted conditionally. This limits performance to 1.64 GB/s. The `emitfn` kernel emits 13 instructions and does little patching, enabling it to encode at 15GB/s.

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

7.3 Automatic Differentiation

Automatic differentiation (AD) computes derivatives of programs by differentiating elementary operations (such as multiplication) and composing those derivatives using the chain rule [5]. It eliminates tedious and error-prone hand-authoring of derivative code, and it gives exact derivatives, unlike finite difference approaches. AD is widely applied to sensitivity analysis and optimization in scientific computing and engineering.

Many applications, such as optimizing an objective function, require the gradient of a single output with respect to multiple parameters, a setting well-suited to *reverse-mode* AD. Reverse-mode AD first runs the program forward, recording each elementary operation and the data needed to compute its derivative on a *tape*. It then interprets the tape backward, accumulating the partial derivative of the output with respect to each intermediate (the intermediate's *adjoint*), terminating with the partial derivatives for the input parameters. Reverse-mode AD can compute arbitrarily many partial derivatives with just two sweeps through the program (one forward and one backward), but the space overhead for the intermediate tape may be significant.

We implemented a reverse-mode AD library with exotypes in Terra, using an approach similar to that of Stan, a C++ library for high-performance statistical inference [25]. Programs written against this library replace floating point numbers with a custom *dual number* type for which arithmetic functions and operators are overloaded. These overloaded functions store their inputs in an object which is placed on an in-memory tape. The reverse pass interprets the tape by calling virtual functions on those objects.

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

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

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

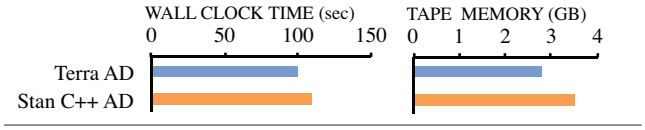

Figure 8. Runtime performance and peak memory usage for reverse-mode AD in Terra and C++. Our Terra implementation achieves comparable speeds and a 25% smaller memory footprint.

layout. Our approach helps alleviate the memory overhead that is the main drawback of reverse mode AD.

We evaluate the runtime and memory performance of Terra AD and Stan C++ AD on a standard optimization task from machine learning: maximum-likelihood training of a logistic regression classifier for hand-written digits using the MNIST dataset [18]. For each implementation, we calculate the gradient of data log probability with respect to model parameters using 6000 data points, and we run 100 iterations of gradient descent. Results are shown in Figure 8. The Terra code achieves runtime performance comparable to C++ with 25% less peak tape memory usage.

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

7.4 Probabilistic Programming

Probabilistic programming languages (PPLs) are a general-purpose modeling tool for artificial intelligence, machine learning, and statistics [9, 16]. Probabilistic programs define probability distributions: the program makes random choices (such as flipping a weighted coin), and running the program produces a sample from the marginal distribution implied by those choices. By conditioning the output of the program on a predicate, programmers can pose interesting queries of their models.

A *universal* PPL such as the probabilistic Scheme dialect Church can describe any Turing-complete, stochastic process, including recursive processes and distributions with infinite support [9]. These languages are expressive, but their implementations are slower than equivalent hand-coded models. One reason for the performance gap is algorithmic: inference algorithms (i.e. implementations of conditioning semantics) must be general-purpose and cannot easily optimize for the statistical traits of specific models. Another reason is computational: probabilistic inference is a numerically-intensive task, but existing universal PPL implementations are high-level and dynamically-typed, making it harder to control their performance.

Addressing this second problem, we implemented a Church-style universal PPL as a library in Terra using exotypes. The query "What is the chance that a patient has lung cancer, given that she has a cough?" in a simplistic medical diagnosis model is expressed in our Terra-based language as:

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

We will highlight two major components built using exotypes and describe why they would be difficult to implement in a low-level language without exotypes, such as C++.

First, the Markov Chain Monte Carlo (MCMC) inference algorithm used to sample from conditioned programs requires that every random choice in the program be given an *address* that is uniquely determined by the choice's structural position in program execution traces [31]. Typically, addresses are managed with a global stack of function callsites: every function call pushes a unique ID on entry and pops the stack on exit. When a random choice is invoked, the sequence of IDs on the stack, plus the number of times that sequence has occurred in the current program execution, uniquely identifies the random choice.

We implement this behavior through multi-stage programming with exotypes. Terra functions are replaced with instances of an exotype `Pfn`, whose `__apply` metamethod wraps the function with code to manage the address stack:

```
id = 0
function pfn(fn)
    -- Exotype declaration
    local Pfn = terralib.types.newstruct()
    -- Wrap function call with address stack code
    Pfn.metamethods.__apply = macro(function(self, ...)
        id = id + 1
        local args = terralib.newlist({...})
        local argIntermediates = args:map(
            function(a) return symbol(a:gettype()) end)
        return quote
            var [argIntermediates] = [args]
            callsiteStack:push(id)
            var result = fn([argIntermediates])
            callsiteStack:pop()
        in result end
    end)
    return terralib.new(Pfn)
end
```

Critically, `__apply` is a macro executed at compile time, so each *callsite* of the function receives a distinct ID.

Managing random choice addresses is more complicated in non-staged languages without exotypes. Previous implementations of universal PPL use either a custom interpreter or a source-to-source transformation [9, 31]. Interpretation is too slow for our performance goals, and source transformation requires a complete parse of the program, which is a steep price to pay for embedding universal PPL in most low-level languages.

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

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

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

Because exotypes allow reflection on functions to drive the generation of new types, we can abstract the details of the random choice as a single library routine. The user only needs to provide three functions, which can be overloaded to handle random choices taking different parameter types. Without exotypes, the behavior of every random choice would be implemented independently for each set of parameter types. This approach would result in redundant code and is more error-prone.

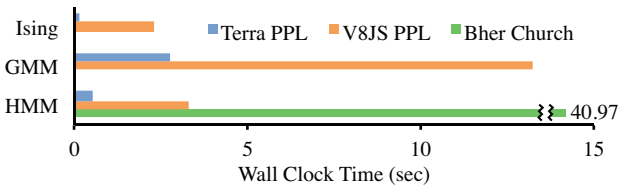

Figure 9. Performance of different probabilistic programming languages on three example tasks. Javascript runs an order of magnitude faster than Bher, and our Terra implementation runs an order of magnitude faster than that.

We evaluate the performance of our Terra PPL implementation on three example programs: conditioned sampling of a length-100 sequence from a Hidden Markov Model with 9 hidden states and 10 possible observations (HMM); learning the parameters of a three-component Gaussian Mixture Model from 1000 data points (GMM); and simulating a 1000-site one-dimensional Ising model (Ising). All examples were run for 5000 iterations of MCMC. We compare performance to that of Bher (a compiled implementation of Church [31]) and a probabilistic dialect of V8 Javascript actively used in teaching. We only evaluate Church on the first example because the other two require language primitives not available in Church.

Results are shown in Figure 9. The Javascript implementation is an order of magnitude faster than Bher on the HMM example. This difference is due in part to V8's optimization engine, and also to the overhead of Bher's purely functional data structures (particularly the trie it uses for random choice addressing). Our Terra implementation is, on average, an additional order of magnitude faster than Javascript. Terra's static typing allows up-front compilation of much more monomorphic code, reducing the number of virtual function calls as well as boxing/unboxing overhead.

Finally, the exotype constructors used in this library are composable with other exotypes. For example, the dual number type from our automatic differentiation library can compose with `RCRecord` to automatically compute derivatives of program probabilities with respect to random choices. We have implemented an MCMC algorithm that uses these derivatives to perform more efficient inference for programs defining mostly-continuous distributions [19].

8. Related Work

Previous work has proposed several optimizations to improve the performance of meta-object protocols in dynamic languages. Most of this work has focused on dynamically-dispatched protocols. For instance, Self, a language based on prototypes that was a precursor to more general protocols, used polymorphic inline caching to optimize method dispatch [13]. Kiczales et al. propose a meta-object protocol for CLOS that uses memoization to speed up method dispatch [15]. These approaches normally still incur some runtime overhead (e.g., to check the cache). In contrast, exotypes are executed during staged compilation, which removes all runtime overhead that is not desired by the user.

There is some work on meta-object protocols that are evaluated statically [3, 17]. The most popular of these approaches provides a meta-object protocol for C++, OpenC++, based on source-to-source translation that allows a programmer to customize the semantics of C++ classes and functions using meta-classes that produce program fragments [3]. Since these methods are applied ahead-of-time, they do not add any runtime overhead, but require that all information used to create the type be available during compilation. In contrast, Terra is a staged programming language, so new exotypes can be introduced during the execution of a Lua-Terra program, and used in newly generated functions.

The original work on multi-stage programming such as MetaML or MetaOCaml focused on generating new code rather than new types [28, 29]. Other systems extend this work to object-oriented languages and provide some degree of staged type computation. Metaphor is a multi-stage language with support for type reflection on a built-in class system [20], and Ur [4] allows first-class computation of records and names based on principles from dependently-typed languages. Both try to guarantee that any code produced using staging will be type correct, making it difficult to generate some types whose semantics depend on dynamically provided data.

Other work on staged programming has focused on optimizations that can be applied to object representations. Rompf et al. propose a system that implements internal compiler passes that use staging to optimize the behavior and the layout of objects in a intermediate representation suited to parallel programming embedded in the Scala language [24]. Type signatures are originally described using Scala's class system, but then their representation and implementation can be optimized through staging. Exotypes additionally allow the original type signatures and behavior to be described programmatically.

Several industrial languages also implement systems similar to exotypes [2, 27]. F# allows *type providers* which can specify types and methods based on external data [27]. Like exotypes, these providers are queried lazily when an operation on the type is requested. However, the goal of type providers is to safely type complex data representations, so providers are normally compiled ahead-of-time rather than during program execution. Furthermore they normally describe types on top of the CLR's object system rather than have the programmer describe their own low-level implementation of types. In the Scala language, the combination of compile-time macros and syntax sugar for supporting dynamic objects allows some types to be described programmatically. Similar to F#, this approach is normally applied ahead-of-time and built on top of the JVM objects rather than a low-level language. Other solutions are tailored to specific problems such as Google's protocol buffers, which provides a language for generating types with serialization behavior in several target languages [11].

9. Discussion and Future Work

We have shown how to supplement a low-level language with a meta-object protocol based on staged programming for describing the behavior and memory layout of types. Exotypes described this way are concise and composable while still allowing the programmer to control their performance. Using these types, we were able to implement libraries for serialization, assembly, differentiation, and probabilistic programming that perform as well as or better than state-of-the-art counterparts while often being substantially more concise. Furthermore, types generated in our examples have simple interfaces similar to existing libraries, encapsulating the use of staged programming inside their implementation. We believe this approach will allow a wider audience to benefit from the performance of staging.

Defining libraries using exotypes has some limitations that can be improved in the future. Dynamically evaluated meta-object protocols like those in Lua can mutate the behavior of objects in existing code by changing the implementation of metamethods. Since Terra is a compiled language focused on high-performance code, we do not allow the results of property queries to change during evaluation if it would require recompilation of Terra code. For instance, the layout of a type provided by `__getentries` cannot change, and methods cannot be modified or removed once they are compiled. However, it is still possible to create new types and methods.

Statically-typed high-level languages generally provide more robust guarantees of type-safety. Lua-Terra programs are dynamically typed, so type errors in exotypes are not reported until a function is compiled. For some uses it is possible to compile all Terra code upfront before evaluating Terra code. In this case, Lua can be viewed as a safer replacement to C++'s preprocessor or template meta-programming that occurs during compilation. We provide the additional flexibility of generating new exotypes at later stages in exchange for complete up-front typechecking.

Furthermore, exotypes are implemented in Terra, a language that allows unsafe type casts, making it possible for programs with poorly written exotypes to crash. It is still possible to implement similar behavior in type-safe runtimes. For instance, F# type providers translate to objects in the CLR, which provides more safety guarantees [27]. However, since type-constructors and other higher-kinded types are implemented in a full language, it is still not possible to guarantee that they will work for all intended inputs. In the future, we want to investigate ways of constraining the properties that a type constructor can request to provide better guarantees of type-safety for all inputs.

Exotypes allow the optimization of individual objects. However, optimizations across objects such as optimizing linear algebra equations on matrices and vectors are more difficult to express. We would like to integrate rewrite-based systems such as the one suggested by Rompf et al. [24] by adding additional meta-methods that are called when an exotype is found in an expression. Integrated with common compiler optimizations, we think these additional rewrites can provide optimization across expressions of objects.

Finally, our examples show that by making libraries more concise using exotypes, it becomes feasible to specialize them for particular uses. In the case of serialization or assemblers, this specialization allows fusion optimizations that lead to large speed-ups over state-of-the-art libraries. We believe that this approach of writing concise but aggressively optimized libraries for specialized uses can be extended to more domains. An object-relation mapping can be optimized for particular data layouts. Class systems can be tailored to the specific code reuse and subtyping requirements of a domain such as abstract syntax trees or graphs. In the future we hope that more patterns of code reuse and optimization can be distributed as concise exotype libraries.

Acknowledgments This material is based on research supported by the DOE Office of Science ASCR in the ExMatEx and ExaCT Exascale Co-Design Centers; DARPA Contract No. HR0011-11-C-0007 and agreement No. FA8750-14-2-0009; 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.

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, 2007.

[2] E. Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *SCALA*, 2013.

[3] S. Chiba. A metaobject protocol for c++. In *OOPSLA*, 1995.

[4] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *PLDI*, 2010.

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

[6] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL*, 1984.

[7] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In *PLDI*, 2013.

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

[9] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *UAI*, 2008.

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

[11] Google. Protocol buffers. `https://code.google.com/p/protobuf`

[12] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA*, 2002.

[13] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI*, 1994.

[14] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of lua. In *HOPL*, 2007.

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

[16] O. Kiselyov and C. Shan. Embedded probabilistic programming. In *DSL*, 2009.

[17] J. Lamping, G. Kiczales, L. H. Rodriguez, Jr., and E. Ruf. An architecture for an open compiler. In IMSA'92 workshop on reflection and meta-level architectures, 1992.

[18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proc. of the IEEE*, 1998.

[19] R. Neal. MCMC using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, CRC Press, London, 2011.

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

[21] P. Norvig. Design patterns in dynamic programming. In *Object World*, 1996.

[22] Oracle. Java object serialization spec. `http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html`.

[23] M. Pall. The LuaJIT project. `http://luajit.org`.

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

[25] Stan Development Team. Stan: A C++ library for probability and sampling. `http://mc-stan.org/`.

[26] N. Sweet. Kryo: Fast, efficient Java serialization and cloning. `https://code.google.com/p/kryo`

[27] 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. `http://research.microsoft.com/apps/pubs/?id=173076`.

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

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

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

[31] D. Wingate, A. Stuhlmüller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *AISTATS*, 2011.