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

Zachary DeVito[1] James Hegarty[1] Alex Aiken[1] Pat Hanrahan[1] Jan Vitek[2]

[1]Stanford University [2]Purdue University

February 14, 2023

Goals

- Performance matters!

Goals

- Performance matters!
- Low-level languages(e.g. C) are good: we need to make best use of features of the target architecture(e.g. vector instructions).

Goals

- Performance matters!
- Low-level languages(e.g. C) are good: we need to make best use of features of the target architecture(e.g. vector instructions).
- Programming is difficult!

Goals

- Performance matters!
- Low-level languages(e.g. C) are good: we need to make best use of features of the target architecture(e.g. vector instructions).
- Programming is difficult!
- Solution: use high-level languages to generate low-level languages code(e.g. FFTW: OCaml $\rightarrow$ C).

New Problems

- In this case, we get three components.

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.
- Compiler: generate taget code based on the plan.

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.
- Compiler: generate taget code based on the plan.
- Runtime: support the generated code and provide feedback to the optimizer.

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.
- Compiler: generate taget code based on the plan.
- Runtime: support the generated code and provide feedback to the optimizer.
- Problem1: How can we get the runtime statistics in the compiler and generate high-performance code dynamically?

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.
- Compiler: generate taget code based on the plan.
- Runtime: support the generated code and provide feedback to the optimizer.
- Problem1: How can we get the runtime statistics in the compiler and generate high-performance code dynamically?
- Problem2: How can we re-use legacy libraries?

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manumal mm.

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manumal mm.
- Use Lua to manipulate Terra code.

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manumal mm.
- Use Lua to manipulate Terra code.
- Shared lexical scoping, which is hygienic.

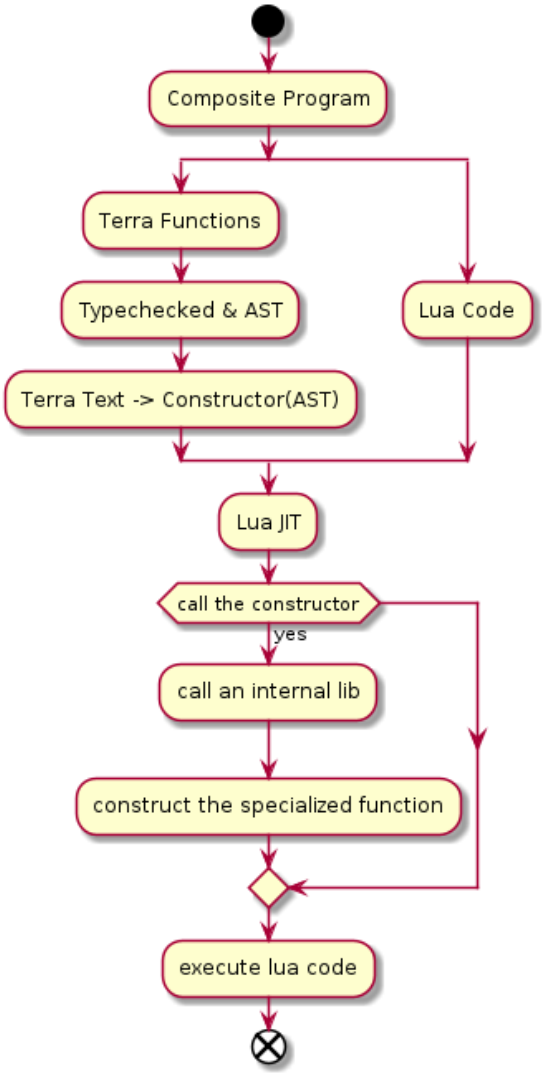
Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manumal mm.
- Use Lua to manipulate Terra code.
- Shared lexical scoping, which is hygienic.
- Terra code runs independently, to avoid including high-level features.

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manumal mm.
- Use Lua to manipulate Terra code.
- Shared lexical scoping, which is hygienic.
- Terra code runs independently, to avoid including high-level features.
- Lua's stack-based C API makes it easy to interface with legacy code.

Two-Language Design



Some Code Examples

```
terra min(a: int, b: int): int
  if a < b then return a
  else return b end
end
struct GreyScaleImage {
  data: &float
  N: int
}
```

Features

- Terra entities are all first-class Lua values.

Features

- Terra entities are all first-class Lua values.
- Terra functions will be executed in LLVM JIT.

Features

- Terra entities are all first-class Lua values.
- Terra functions will be executed in LLVM JIT.
- You can dump Terra functions to an object file(i.e. something.o in Linux) if you like.

Features

- Terra entities are all first-class Lua values.
- Terra functions will be executed in LLVM JIT.
- You can dump Terra functions to an object file(i.e. something.o in Linux) if you like.
- Quotation: using brackets([]) for escaping and backtick(expressions)/quote keyword(statements) for creating quotation.

Quotation Example

```
local a = 5
terra sin5()
  return [ math.sin(a) ]
  end
function addtwo(a,b)
  return `a + b
end
local printtwice = quote
  C.printf("hello\n")
  C.printf("hello\n")
end
```

It Just Works!

```
-- C++                          | -- Lua/Terra
int add(int a, int b) {         | terra add(a : int,b : int) : int
    return a + b;               |     return a + b
}                               | end
                                |
                                | -- Conditional compilation is done
                                | -- with  control-flow that
                                | -- determines what code is defined
#ifdef _WIN32                   | if iswindows() then
void waitatend() {              |     terra waitatend()
    getchar();                  |         C.getchar()
}                               |     end
#else                           | else
void waitatend() {}             |     terra waitatend() end
#endif                          | end
                                |
                                | -- Templates become Lua functions
                                | -- that take a terra type T and
                                | -- use it to generate new types
                                | -- and code
template<class T>               | function Array(T)
struct Array {                  |     struct Array {
    int N;                      |         N : int
    T* data;                    |         data : &T
                                |     }
    T get(int i) {              |     terra Array:get(i : int)
        return data[i];         |         return self.data[i]
    }                           |     end
                                |     return Array
};                              | end
typedef                         |
Array<float> FloatArray;        | FloatArray = Array(float)
```

It Just Works!

- Now we can generate code dynamically.

It Just Works!

- Now we can generate code dynamically.
- e.g. block the loop nests to make the memory access more friendly to the cache.

The Formal Calculus: Terra Core

Summary

- Two-Languages design: Lua + Terra.

Summary

- Two-Languages design: Lua + Terra.
- Shared lexical scoping.

Summary

- Two-Languages design: Lua + Terra.
- Shared lexical scoping.
- Seperate Evaluationn: Lua(LuaJIT), Terra(LLVM JIT).

Summary

- Two-Languages design: Lua + Terra.
- Shared lexical scoping.
- Seperate Evaluationn: Lua(LuaJIT), Terra(LLVM JIT).
- Type Reflection.

Advantages

- No need to write C, but the performance is still good.

Advantages

- No need to write C, but the performance is still good.
- Generating code dynamically allows us to use runtime information from LuaJIT.

Advantages

- No need to write C, but the performance is still good.
- Generating code dynamically allows us to use runtime information from LuaJIT.
- Easy to re-use C/C++ libraries in Lua.

Advantages

- No need to write C, but the performance is still good.
- Generating code dynamically allows us to use runtime information from LuaJIT.
- Easy to re-use C/C++ libraries in Lua.

Shortages

- Lua is not statically typed.

Shortages

- Lua is not statically typed.
- LLVM + Lua is compromise, because we don't want to re-implement the whole LuaJIT!