NAME_TBD- A Domain Specific Language for Implementing Finite State Machines in Hardware

Leonard Truong
lenny@stanford.edu

James Hegarty
jhegarty@stanford.edu

Ross Daly
ross.daly@stanford.edu

Pat Hanrahan
hanrahan@cs.stanford.edu

Abstract

1 Introduction

We propose a domain specific language for implementing finite state machines in hardware. The language is embedded in C to provide seamless interoperability with existing codebases in any language TODO: Well any language with a good FFI could make this argument, probably need to reword this to avoid religious arguments. The user is provided with a set of operators, types, and built-in functions that are described in detail in Section 3. The key language extension is the introduction of the `yield` keyword. This allows users to implement finite state machines using the coroutine programming model (described in detail in Section 1.2.2). The compiler, described in Section 4, analyzes the program to construct an abstract representation of the state machine, which is then lowered into CoreIR.

A side-goal of this project is to construct a lightweight toolkit for embedding DSLs in C. While embedding a language in C is likely to introduce some headaches for the toolkit (i.e. we can't just hook into the built-in parser like in Lua or Python), we see this as a future-proof host language choice. TODO: That is, C isn't going anywhere, so at least the host language won't become obsolete.

1.1 Motivation

A hardware implementation of a finite state machine would traditionally be written in a hardware description language (HDL) such as Verilog. In practice, this partially reduces to a giant `case` statement that enumerates all the possible state transitions. This approach suffers from verbosity and is prone to errors. TODO: Probably good to have James H. (and Ross?) to write something here since they've actually dealt with this

An alternative approach would be to write the state machine in C and use a high-level synthesis tool (HLS) to automatically construct a hardware implementation. TODO: Argument here is basically general purpose versus DSL compilers, that is, HLS=general purpose compiler Being able to describe the program in a higher-level, imperative language is convenient for the programmer, but makes the job difficult for the compiler. For example, typical HLS systems decouple the behavior of the program from clock-level timing, which can make it difficult to understand the semantics of how operations are executed in hardware (sequential or parallel). This creates issues in designing and debugging implementations.

1.2 Background

1.2.1 Automatons and Finite State Machines

Automatons are abstract models of machines that perform computations on an input by moving through a series of states or configurations. A finite state machine (FSM) is an automaton in which the state set contains only a finite number of elements [1].

Two major types of finite state machines are the *Mealy* machine and *Moore* machine. A Mealy machine determines its output as a function of its current state and input, while a Moore machine's output is based purely on the current state.

1.2.2 FSMs as Co-routines

TODO: Formally define co-routines By definition, co-routines preserve the state of the function between calls, making them a viable programming model for finite state machines.

1.2.3 Hardware Description Languages

1.2.4 High-level Synthesis

2 Related Work

2.1 greenery

greenery [2] is a set of tools for parsing and manipulating regular expressions (greenery.lego), for producing finite-state machines (greenery.fsm), and for freely converting between the two. The main feature of greenery is the ability to convert two regexes into finite state machines, computing the intersection of the two FSMs as a third FSM, and converting the third FSM back to a regex. TODO: This library might be useful if we can find patterns where the intersection of FSMs would allow us to reuse hardware, seems like a stretch goal though

3 Language Specification

The language was designed to closely match the syntax of the host language (C) to reduce the cost of learning and the burden of context switching between languages. Our only extension of the C-language is the yield keyword to indicate the end of a clock cycle. We modify the semantics of certain C operators to better suit hardware construction. Finally, we restrict the use of valid C syntax that is not necessary for the functionality of our language.

3.1 Grammar

```
<stmt>   ::= <expr>  # expr
           | <sym> <sym>  # variable declaration
           | <sym> = <expr>  # assignment
           | <sym> <sym> = <expr>  # declaration with assignment
           | <sym> <sym>(<sym>, ..., <sym>) { <stmt>* } # function declaration
           | if (<expr>) { <stmt>* }  # if then
           | if (<expr>) { <stmt>* } else { <stmt>* }  # if then else
           | while (<expr>) { <stmt>* }  # while loop
           | for (<expr>; <expr>; <expr>) { <stmt>*}  # for loop
<expr>   ::= null
           | <number>
           | <sym>
           | (<expr>)
           | <expresson> <binop> <expr>
           | <unop> <expr>
           | <sym>(<expr>, ..., <expr>)  # function call
           | yield
           | return <expr>
```

```
          # TODO: Do we need something like reg[2:0]
<sym>    ::= /[a-zA-Z_][a-zA-Z_0-9]*/
<number> ::= /[0-9]+/
<binop>  ::= !, &, |, ~&, ~|, ^, ~^, +, -, *, /, %, <<, >>, >, >=, <,
           |  <=, ==, !=, &&, ||
<unop>   ::= ~, +, -, (<sym>)
```

3.2 Types

3.2.1 input

The `input` type is used to describe inputs to the state machine. The value of an input variable is read-only.

3.2.2 output

The `output` type is used to describe outputs to the state machine. The must be set during the first clock cycle (cannot be undefined) TODO: Should we instead initialize it to some value?. An output type must only be written once during a single clock cycle.

3.2.3 reg

The `reg` type is used to describe a register for implementing sequential logic. An instance of `reg` will retain it's value till the next clock cycle. An instance of `reg` can be read from freely, but only written to once during a clock cycle.

3.2.4 wire

The `wire` type is used for implementing combinational logic. They can be read freely but only assigned once per clock cycle. They must be driven by a continuous assign statement or from a port of a module.

3.3 yield

3.4 Example Programs

3.4.1 Downsample module

```
void downsample(input A, output B, output valid) {
  for (int y = 0; y < SIZE_Y; ++y) {
    for (int x = 0; x < SIZE_X; ++x) {
      valid = (x % 2);
      B = A;
      yield;
    }
  }
}
// TODO: Need to decide on how this would be wrapped
// example:
//  setup A, B, valid
//  while (true) {
//    downsample(A, B, valid)
//  }
```

3.4.2 VGA Controller

```
bool column_in_display(x) {
  return x >= VGA_X_MIN && x < VGA_X_MAX;
}
bool row_in_display(y) {
```

```
  return y >= VGA_Y_MIN && y < VGA_Y_MAX;
}
void vga_controller(input A, output B, output valid) {
  for (int y = 0; y < size_y; y++) {
    for (int x = 0; x < size_x; x++) {
      valid = column_in_display(x) && row_in_display(y);
      B = A;
      yield;
    }
  }
}
```

4 Language Internals (Compiler)

4.1 Parser

Option 1: Use a tool like Bison to generate a parser from the above grammar.

Option 2: Create something better than Bison (lightweight, intuitive, simple, verifiable?)

4.2 Internal Representation of State Machines

The next phase begins by converting the programs AST into a control flow graph, giving us all the possible paths that the program can take during execution. We can then observe all the paths between yield statements as possible state transitions during a clock-cycle. With these paths we can also perform type checking (i.e. ensure that the outputs are only written once in a clock cycle).

4.3 Lowering to CoreIR

5 Milestones

- Proposal
- Parser
- Control flow graph analysis TODO: Is this what you call constructing the CFG?
- Infer transition set (paths between yields)
- Infer FSM state (function state)
- Implement a way to verify the internal representation of an FSM (programmatic interface or visual/graphical representation to check manually)
- Type check paths between yields
- Lower transitions and FSM state to CoreIR logic

6 Conclusion

Acknowledgments

References

References

[1] Basics of automata theory. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html`. Accessed: 2017-01-18.

[2] ferno. greenery. `https://github.com/ferno/greenery`, 2017.