Final Report - OCaml Act - Henry Heffan

Abstract

Professor Manohar's lab develops a toolchain that enables the design of asynchronous circuits. Currently, users interface with these tools using a new programming language (called “Act”) developed by the lab. However, Act has a number of problems, both for people designing chips and people developing the toolchain. To address these problems, this project presents a new way to interface with the tools. A user can write code, not using the Act language, but instead using a library embedded inside of OCaml, an existing language. The library leverages existing OCaml tools to require very little code to implement. This provides a number of benefits, both for chip designers and tool writers. For chip developers, it removes a number of sharp edges that currently exist in the language, provides better compiler time code generation, provides code formatting and syntax highlighting, and provides access to better testing frameworks. Moreover, toolchain developers can exploit the OCaml compiler to remove the vast majority of the front-end of the Act compiler, along with allowing new tools to be written using OCaml instead of C.

Background - What is a computer chip? How does a circuit “compute” something? Why use a programming language to design circuits? Why a new programming language?

Note that this section is intentionally simplifying how computer chips work and are designed. It is meant to be a useful approximation of the truth, in that it should explain the motivation of why my tool is needed and the problems that it attempts to solve.

Computer chips are circuits: They consist of wires and transistors. But, the circuit components are microscopic. The transistors in modern chips can be as small as 5 nm across. So, modern computer chips can contain billions of gates and wires. This allows one chip to contain an incredibly complex circuit. One can design circuits to carry out computations, and more complex circuits can carry out more complex computations. The ability to manufacture circuits with such small components is why modern chips are so powerful. A user designs a circuit by producing a diagram specifying where wires and transistors should go. A manufacturer can then produce a physical chip from that diagram.

Information gets carried around a computer chip by the voltages on wires. Wires can have any voltage in a continuous range, but computer chip designers often use the voltages to encode binary values. They use a “high” and “low” voltage to encode “zero” and “one” respectively. When the inputs are encoded like this, transistors act like logical “nand” and “nor” gates. But, if any input wire is somewhere in between “high” and “low” voltage, then the transistor will have much more complex behavior. Two different transistors might even react differently to the same intermediate voltage. One transistor might interpret a voltage as a “zero” and the other might interpret that same voltage as a “one”. If this happens to the wrong pair of transistors, it might

cause a short circuit and destroy the chip! Thus, engineers often design circuits to avoid these intermediate voltages. They design each component on the chip assuming its inputs will never be these intermediate values, and (try to) ensure that each component never produces such outputs.

Unfortunately, any time a wire changes voltage (e.g. from “high” to “low”), it *must* pass through the intermediate voltages (since voltage is a continuous function of time). If not designed correctly, temporary intermediate voltages in one part of the circuit might trigger a chain reaction and cause permanent intermediate voltages in a different part of the circuit.
It is really hard to check a general circuit to ensure that intermediate voltages will never end up propagating across the chip. One common strategy to protect against this is to add a clock into the circuit. The circuit is designed in “chunks”, where each chunk receives inputs at the beginning of a “clock cycle” and will have its outputs read at the end of a “clock cycle.” Each chunk is then designed so that it will never have invalid values on its outputs at the end of a clock cycle. If done correctly, this stops intermediate voltages from propagating across the chip.

Even so, designing a circuit by hand that does what you want is really hard. As I said before, modern chips can have billions of wires. It would be extremely time-intensive to design them all manually. Moreover, a design specifying the desired location of wires carries no semantic information. So, if the circuit does not work as expected, it is extremely difficult to figure out what went wrong. Third, even though there are good principles of how one *should* design a circuit, checking that the circuit respects these principles is very difficult. So, instead of designing circuits by hand, people often use a programming language to explain *what* they want their circuit to do (and possibly how to do it). They then run a compiler to convert their program into a diagram of a circuit. The most common programming language for chip design is called Verilog. It is particularly good at designing circuits which include clocks.

C was designed to map really nicely down to assembly. So, a C program might be more readable than assembly, but it should have similar performance (if the compiler does a good job) to writing assembly by hand. Often, it will even do a better job than a person, because the compiler has a better understanding of how fast different things are. It can do this for two reasons. First, it provides abstractions that map directly to assembly. For example, addition in C maps directly to the “add” instruction in most assembly languages. Second, C strategically makes certain behavior “undefined”, which makes space for the optimizer to select among different ways to implement a given piece of code. In any case, it is easier to write a correct program in C than assembly. This is because C allows a user to express semantic meaning, and since C restricts the ability of a programmer to do the incorrect thing.

Although people have tried to build tools that allow one to design a circuit in C, this has not been very successful because C does not map well onto hardware primitives. Verilog is used so widely precisely because it maps to circuits well.

Professor Manohar's lab works on tools to design circuits that don't include clocks (called "asynchronous circuits"). One of these tools is a compiler for a new language created by the lab. A new language is needed because Verilog does not have good abstractions for specifying the behavior of asynchronous circuits. This new language (called "Act") allows chip designers to express their intent directly (making it easier to write correct circuits) and allows for optimizations that wouldn't be possible in Verilog (allowing for the compiler to output faster circuits).

For my senior project, I built a new front end for the compiler. The front end of a compiler is the part that takes in the source code, checks that the source code is correct, and outputs a simplified version of the same code (called an "intermediate representation") that is suitable for optimization. This provides many advantages over the current front-end to the compiler, which I will explain below. I also built a number of optimizations in the new compiler and wrote a simulator for Act code.

Overview of the current Act language and compiler, and motivation for the ocaml_act library.

The Act language consists of one "frame" language and several "sublanguages". A program in Act consists of a collection of "processes" (which do computations and produce outputs), and "channels" (which carry values between processes). A process can contain subprocesses and specify how channels connect them. A process can also use a sublanguage to directly express a computation that reads values from channels and produces outputs on some other channels. Some of the sublanguages are high-level (kind of like C), and some of the sublanguages are low-level (kind of like assembly). The frame language provides several "compile-time code generation" features. It supports C++ style templates which can be instantiated with different integer values and supports "compile time loops" (which are loops of fixed length unrolled at compile time).

The way a compiler generally works is that it takes a text file, and then does a bunch of computation to turn the text into a tree representing the program. This tree represents the structure of the language in the file. Along the way, the compiler checks that the text file contains a valid program. This involves checking a lot of things. For example, that there are no illegal characters in the file (while lexing), that all pairs of parentheses are balanced (while parsing), that every variable which is used is declared (semantic analysis), and that strings are never assigned to ints (type checking). And, if things go wrong, the compiler must issue a helpful error message, so that the programmer knows what they must fix. These steps together are called the "front-end" of a compiler. A compiler then (usually) transforms this abstract syntax tree into a

simplified “intermediate representation”. This involves expanding out all the macros and “syntactic sugar”. Finally, the “back-end” of the compiler then does optimizations on the intermediate representation and then converts the optimized intermediate representation into the output language.

Writing either a compiler front-end or compiler back-end is complex. My project focused (mostly) on replacing the front-end of the current Act compiler. The current front-end has a number of weaknesses. Some of these affect users of the Act language and some affect tool developers.

The current design of the Act language has features that make it hard for chip designers to use. First, Act has several “compile-time code generation” features (described above), but they are somewhat ad-hoc. Each feature can only be used in certain contexts, and compile time code does not allow for “dynamic” manipulation of arrays. When I took Silicon Computation, I wrote a Python script to generate Act code for my final project, because that was more versatile than using the language features. Second, Act has a number of sharp edges regarding the “width” of integers. For example, 100 + (-3) is equal to 105, which is probably unexpected. Even worse, the behavior of “runtime” and “compile time” integers are not the same! Third, the Act language overloads the “bool” type to represent both a “wire” in the “low-level” sublanguage and a “1-bit wide integer resulting from comparison” in the “high-level” sublanguage. These are semantically different but can be passed between one another. Fourth, one can pass flags to the compiler that change the semantics of a function. This means that one can not simply compose two programs, but must check how each program is supposed to be compiled. Fifth, there currently is no good “tooling support” for the Act language; There is no IDE support, no syntax highlighting, no Act-code formatter, and no testing infrastructure. It would be possible to build this, but would take a significant amount of effort to both build and maintain.

The current design of the Act compiler has features that make it hard for developers to work on it. First, it is written in C. Since C doesn't have a garbage collector, the compiler has a lot of code to deal with memory allocation and deallocation, which is easy to mess up when building new tools or editing existing tools. Second, C has a weak type system, so the compiler does a poor job checking that a given program is correct before compiling it. A large percentage of development time is spent chasing down bugs that the compiler of a higher-level language would be able to catch. Third, C doesn't have good support for “coproduct types,” so changing the abstract syntax tree of the intermediate representation of the compiler is very difficult to do correctly. It is easy to change the struct definitions, but very hard to find every location that uses the changed struct. Fourth, the lexer and parser are built using a custom code generation tool (which employs a lot of macros), and Act has a very complex grammar. Currently, the front-end in the Act compiler includes more than 50,000 lines of C code! This makes it very hard to alter. Moreover, the

custom code generator has a known bug where it is unable to parse large files, but there is not (or at least I could not find) an easy fix.

My project was to build a new front-end for the compiler and a new revision of the language with the goal of addressing these problems. I decided to embed the Act language inside of an existing language. By this, I mean that instead of writing Act code in its own kind of file, one would build up Act code as a data structure within a different language. For illustration, this is what I ended up with, but I will explain how this works in more detail in the "Examples" section.

```
5  let split ~dtype i1 o1 o2 =
6    let var1 = Var.create dtype in
7    let b1 = Var.create CBool.dtype ~init:false in
8    Chp.loop
9      [
10       Chp.read i1 var1;
11       Chp.if_else
12         Expr.(var b1)
13         [ Chp.send_var o1 var1 ]
14         [ Chp.send_var o2 var1 ];
15       CBool.Chp.toggle b1;
16     ]
17   |> Process.of_chp ~iports:[ i1.u ] ~oports:[ o1.u; o2.u ]
```

The idea of building a new programming language inside an existing language was inspired by two existing libraries. First, there is a C++ library called Halide. It is a language for writing high-performance image-processing code. Instead of developing a whole new syntax, one builds a Halide program by using classes in the library within a C++ program. When that program is run, it generates a library that has the result of compiling the Halide code. This allowed them to avoid having a separate front-end for the compiler. Also, it allowed them to provide a "Just in time compiling" mode within C++. A second library that inspired this project is the Hardcaml Library. It is a library, written in OCaml, which wraps around Verilog. It allows one to write Verilog code in OCaml and removes some of the sharp edges of Verilog without hurting the performance of the resulting code. It also allows testing to happen from OCaml. This library allows its users to benefit from OCaml's better testing and development infrastructure and tooling.

Embedding the Act inside of an existing language has a bunch of advantages for users of the Act language. First, the existing user gets a bunch of tooling for free. Any tools which work on the existing language automatically work on the embedded version of Act, since the embedded version of Act is just a subset of the existing language. Second, redesigning the front-end of Act allows us to remove many of the sharp corners in the existing version of the Act language. So, users are less likely to shoot themselves in the foot. If the existing language has good support for

inline tests, then we can write a simulator that will leverage that to get good support for inline tests for Act. This will make it easier for a developer to ensure the correctness of their code. We could in the future also add a way to use inline tests to prove circuits correct using proof systems. Finally, developers will be able to leverage the full expressive power of the existing language to do “compile time generation” of code. This will be much more powerful than the existing “compile-time code generation” options and will avoid the need to write programs to generate Act programs.

Moreover, embedding the Act language inside an existing language has a bunch of advantages for developers. First, we can leverage the existing languages compiler to do most of the front-end work for us. It will lex, parse, and do semantic analysis for free. This means we can replace 50 thousand complex lines of code with 5 thousand simple lines of code, which will be much easier to change and maintain. Second, if the existing language already has good developer tooling, we will not need to build IDE support for the Act language ourselves. Second, we will get to write our program in whichever existing languages we pick, instead of in C. If we pick a language that is easier to write than C, this will speed up future development of the compiler. Third, it will make it easy to write future tools in whichever existing language we pick. So, if we pick a language that will allow faster development than C, this will have an outsized effect on the total work that gets done in the lab. Moreover, if the library I develop provides a useful intermediate representation, it will be far easier to use it as a front-end for other Act tools.

I chose to use OCaml as the existing language for a few reasons. First, it has a garbage collector. This makes it much faster to write the compiler, but also is a benefit to users of the library. Otherwise, they would need to manually manage the memory in the data structures representing their program as they built them up. In fact, I tried writing my library in Rust first but was unable to come up with a nice interface for my library. Second, OCaml has algebraic data types. These make writing a compiler much easier, as they make it possible to represent the parse tree much more naturally. Third, OCaml has support for first-class functions, which are used heavily throughout the interface to the program. Fourth, OCaml already has good tooling, so we will be able to leverage its IDE support and testing framework and code formatting and syntax highlighting tools for free. Finally, OCaml has a very expressive type system, which makes it possible to embed the Act type system within OCaml. This allows us to leverage the OCaml type checker to do most of our type-checking, saving us a lot of code and complexity.

Examples

First, here is an example of how to build a simple buffer using the ocaml_act library. The first function, when called, instantiates a circuit implementing a buffer. The second block of code instantiates a buffer and then runs a test to check that the buffer behaves as expected.

```
(* This function generated the IR for a simple buffer. It then returns the IR,
   along with the "write end" of the input channel and the "read end" of the
   output channel. *)
let simple_buffer () =
  let i = Chan.create CInt.dtype_32 in
  let x = Var.create CInt.dtype_32 in
  let o = Chan.create CInt.dtype_32 in
  let chp = Chp.loop [ Chp.read i.r x; Chp.send_var o.w x ] in
  (chp, i.w, o.r)

let%expect_test "test" =
  (* Instantiate the buffer *)
  let chp, i, o = simple_buffer () in
  (* create a simulation *)
  let sim =
    Sim.simulate_chp chp ~user_sendable_ports:[ i.u ]
      ~user_readable_ports:[ o.u ]
  in

  (* Set up a simulation step. We will send the value `3` on `i`, and check that
     we can read a `3` on `o`. *)
  Sim.send sim i CInt.three;
  Sim.read sim o CInt.three;

  (* Then, wait for all the queued action to complete. If anything goes wrong,
     or if any actions dont compelte, print an error *)
  print_s [%sexp (Sim.wait sim () : unit Or_error.t)];

  (* This contains the expected contences of stdout after running the test. See
     the section on expect_tests in
     https://dev.realworldocaml.org/testing.html. *)
  [%expect {| (Ok ()) |}]
(* $MDX part-end *)

```

This second example demonstrates how to build a more complex “tree” buffer. One way to build a buffer with a given capacity is to chain several simple buffers end to end. Instead, here we build a tree of splits and merges. This can be preferable because the number of copies needed to get a value from the input to the output of a tree buffer is only logarithmic in the capacity of the buffer, rather than linear, as it is in the chained simple buffer case.

```
let split ~dtype i1 o1 o2 =
  let var1 = Var.create dtype in
  let b1 = Var.create CBool.dtype ~init:false in
  Chp.loop
    [
      Chp.read i1 var1;
      Chp.if_else
        Expr.(var b1)
        [ Chp.send_var o1 var1 ]
        [ Chp.send_var o2 var1 ];
      CBool.Chp.toggle b1;
    ]
  |> Process.of_chp ~iports:[ i1.u ] ~oports:[ o1.u; o2.u ]

let merge ~dtype i1 i2 o1 =
  let var1 = Var.create dtype in
  let b1 = Var.create CBool.dtype ~init:false in
  Chp.loop
    [
      Chp.if_else Expr.(var b1) [ Chp.read i1 var1 ] [ Chp.read i2 var1 ];
      Chp.send o1 Expr.(var var1);
      CBool.Chp.toggle b1;
    ]
  |> Process.of_chp ~iports:[ i1.u; i2.u ] ~oports:[ o1.u ]

let rec buff ~depth ~dtype i1 o1 =
  (if depth <= 0 then failwith "depth too low"
   else if Int.equal depth 1 then
     let chan1 = Chan.create dtype in
     let chan2 = Chan.create dtype in
     [ split ~dtype i1 chan1.w chan2.w; merge ~dtype chan1.r chan2.r o1 ]
   else
     let chan1a = Chan.create dtype in
     let chan1b = Chan.create dtype in
     let chan2a = Chan.create dtype in
     let chan2b = Chan.create dtype in

     [
       split ~dtype i1 chan1a.w chan2a.w;
       buff ~dtype ~depth:(depth - 1) chan1a.r chan1b.w;
       buff ~dtype ~depth:(depth - 1) chan2a.r chan2b.w;
       merge ~dtype chan1b.r chan2b.r o1;
     ])
  |> Process.of_procs ~iports:[ i1.u ] ~oports:[ o1.u ]

let buff ~depth ~dtype =
  let i = Chan.create dtype in
  let o = Chan.create dtype in
  let top_process = buff ~depth ~dtype i.r o.w in
  (top_process, i.w, o.r)
```

This code contains four functions, each of which instantiates a circuit when called. Notice that the third function makes calls to the first two functions, and also is recursive. The fourth function makes use of OCaml to provide a nicer wrapper around the third function, masking some of the unneeded parameters. Then, the following code runs tests on the fourth function, checking that it produced a circuit that functions correctly.

```
56  let%expect_test "test depth 3" =
57    let process, i, o = buff ~depth:3 ~dtype:(CInt.dtype ~bits:9) in
58    let sim = Sim.simulate process in
59
60    let l =
61      [ 1; 5; 9; 33; 123; 258; 500; 7; 9; 4; 5 ] |> List.map ~f:CInt.of_int
62    in
63    List.iter l ~f:(fun v -> Sim.send sim i v);
64    Sim.wait' sim ();
65    List.iter l ~f:(fun v -> Sim.read sim o v);
66    Sim.wait' sim ();
67    [%expect {|
68      (Ok ())
69      (Ok ()) |}]
70
71  let%expect_test "test depth 5" =
72    let process, i, o = buff ~depth:5 ~dtype:(CInt.dtype ~bits:1) in
73    let sim = Sim.simulate process in
74
75    let l =
76      [ 1; 0; 0; 0; 0; 1; 1; 0; 1; 1; 1; 0; 1; 1; 0 ] |> List.map ~f:CInt.of_int
77    in
78    List.iter l ~f:(fun v -> Sim.send sim i v);
79    List.iter l ~f:(fun v -> Sim.read sim o v);
80    Sim.wait' sim ();
81    [%expect {|
82      (Ok ())
83      (Ok ()) |}]
```

I have more complex examples, including an example of how one might implement a simple CPU using the ocaml_act library. However, they are too large to include in this paper. They can be found at https://github.com/asyncvlsi/ocaml_act. In particular, you should look inside the "examples" and "test" folders.

Library design

The main complexities in designing this library were (1) encoding the Act type system inside the OCaml type system and (2) providing a good intermediate representation. I solved these problems the following way.

My OCaml library consists of two layers. The "intermediate representation" layer only deals with integers (no boolean, enums, etc.), and has a very limited number of syntactic constructs. For example, instead of supporting both an "if-else" statement and a "select" statement, it just supports one of these. OCaml automatically protects against certain kinds of error (such as passing a variable in place of a list), but it does not protect against all the possible errors a

programmer could make (for example, passing a variable with a width greater than one as the control flag of a select statement).

So, on top of the “intermediate representation” layer, I build a wrapper layer that enforces correct usage of the bottom layer. It does this using “phantom types.” These are types which are exposed in the interface of the module but are not used in its implementation. The OCaml type-checker will use these types when ensuring that the program follows the rules as expected. The idea is that every ocaml_act variable, no matter what type (e.g. an int or a bool), is just an integer under the hood. But, the OCaml type system thinks of it as being variables of different types, and so will print an error if one is used in the wrong location. Moreover, the user interacts with all these types through function calls. These function calls implement additional type checks (such as checking bit widths), helping further ensure that a program is constructed correctly. Behind the abstraction layer, my library immediately builds up the intermediate representation. How this is implemented is interesting, but is better learned by looking at the source code.

One clever thing about the library is how it captures the location in code where an ocaml_act variable/expression was created. This is necessary for printing useful error messages to a user if their program turns out to have a problem with it. The way the library does this is that immediately after any ocaml_act library function is called, the library captures a stack_trace, and uses that to get the caller's location.

In addition to the front-end, I implemented a simulator for Act code, which enables inline testing of OCaml code. This is useful because it allows tests to be tightly coupled with the code they are testing. Currently, tests are implemented using a different language than Act. It also ensures tests are run every time the code is altered (because the OCaml compiler will run them for us), and allows us to use OCaml to generate complex tests.

I also wrote part of the back-end of a compiler, which implements a number of optimizations. These can be run before exporting the Act code. I did this to demonstrate that the full tool-chain could be done just in OCaml. I also did this to demonstrate that writing tools in OCaml should be much easier than writing them in C++. The optimizations were primarily a port of existing C++ code, but it took approximately one-quarter as many lines of code because writing code in Ocaml is much easier than C++.

Conclusion

I think there are two main takeaways from this project.

First, adopting an OCaml front-end in place of the existing front-end would significantly improve the experience of using Act for both developers and chip designers. Further, it would

make it much easier to remove sharp corners from Act and would make future development and maintenance much less expensive.

Second, I think this project demonstrates that building a library like the ocaml_act library inside an existing language is a viable alternative to designing a new domain-specific language. It is much faster to write, and much cheaper to maintain, but allows building as expressive a language as one would have otherwise.