Static Type Systems in the Implementation of Hardware Description Languages

By Sam Cohen, ECE '22

Introduction

Today, VHDL and Verilog are the primary hardware description languages that engineers use to design and simulate digital devices. These languages, however, adopted the role of simulation and design tools far after they were initially developed. VHDL, for example, was invented in the 1980s for *documentation* of digital systems, not their simulation or synthesis [1].

But what made VHDL a good language for documenting digital systems? In this paper, I will argue that a strong, dependent type system is what makes VHDL and Verilog excellent tools not only for simulation and synthesis, but also for documentation of a designer's intent.

Figure 1 shows a VHDL program that describes a 2-bit multiplexer. A multiplexer is like a railroad switch; It allows any number of data inputs to be switched into a single output using a control signal. In the example of Figure 1, 4 data inputs, each 2 bits wide, are switched into a single output (See Figure 2 for a diagram).

Figure 3 shows the semantically relevant portions of the module defined in Figure 1. In essence, the VHDL of Figure 3 specifies that each index of the array `inputs` is connected to one of the inputs of the multiplexer. Secondly, it specifies that the output of the multiplexer, `out`, is the index of `inputs` that corresponds to `s` interpreted as an integer.

If Figure 3 specifies the entire semantics of the design, then why does it represent less than half of the lines required to fully define the multiplexer? Though the condensed listing does explain how data is routed from the input to the output of the device, it does not explain the *shape* of that data. In the context of digital design, this is not only an important consideration, it is the *only* consideration.

Every digital system uses bits to encode information. In a computer, for instance, the memory space is divided into regions for executable code and data. The data regions are composed of words which can contain integers, characters, floating point numbers, or other types of information. If we were to dump the contents of a computer's memory onto a tape, however, and try to extract all the integers or all the characters, it would be impossible.

```
entity mux is
port(
  d0  : in  std_logic_vector(1 downto 0);
  d1  : in  std_logic_vector(1 downto 0);
  d2  : in  std_logic_vector(1 downto 0);
  d3  : in  std_logic_vector(1 downto 0);
  s   : in  std_logic_vector(1 downto 0);
  out : out std_logic_vector(1 downto 0));
end mux;

architecture synth of mux is
type input_array is array (0 to 3) of
    std_logic_vector(1 downto 0);
signal inputs  : input_array;

begin

  inputs(0) <= d0;
  inputs(1) <= d1;
  inputs(2) <= d2;
  inputs(3) <= d3;

  out <= inputs(to_integer(unsigned(s)));

end synth;
```

Figure 1: VHDL describing a 2-bit multiplexer

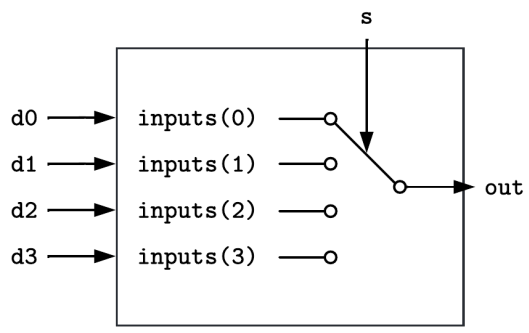

Figure 2: Block diagram of the 2-bit multiplexer

```
begin

  inputs(0) <= d0;
  inputs(1) <= d1;
  inputs(2) <= d2;
  inputs(3) <= d3;

  out <= inputs(to_integer(unsigned(s)));

end synth;
```

Figure 3: Semantically relevant portions of 1

This is because the data inside a digital system contains no intrinsic type information. In other words, the bits that represent integers and strings are exactly the same, and it is up to the computer's internal accounting to decide what bits correspond to integers, strings, executable code, or garbage.

In VHDL or Verilog, all of the underlying datatypes are identical. They are arrays of one or more bits. So, in a sense, the only information that an HDL encodes is type information: information about how those ordinary bits are interpreted within the digital system.

Type systems often come in two flavors: nominal type systems and structural type systems. In a nominal type system, two types, even if identical in shape, are not equal. Only types with the same name are considered equal [2]. At first this seems like a poor choice; two variables that encode integers, if given different type names, would not be considered compatible (even though all operations on those two integers would be defined and predictable).

It makes sense that VHDL and Verliog adopt *nominal* type systems as opposed to structural type systems because the value of types is so amplified in a hardware description language and the distinction between the internal representation of data is so minimal. By assigning a type to a value, the programmer is not only saying what shape the underlying data has, they are asserting how the value should be used.

In the code listing of Figure 1 we see the effects of this type system in the type conversions in the assignment of `out`. The variable `s` has type `std_logic_vector`, but the array `inputs` must be indexed by an integer.

In VHDL, no hardware is actually generated when synthesizing the type conversions seen in Figure 1. The internal representation of logic vectors and unsigned integers are identical. VHDL enforces type conversions like these because types are used to convey *intent*, not only representation.

Types can be so descriptive about the intent of a program that, in some cases, the types are all that are necessary to fully specify an algorithm. In one 2016 paper by Polikarpova, Kuraj, and Solar-Lezama, it was demonstrated that polymorphic refinement types were often sufficient to fully specify a recursive function for a large class of algorithms [3]. In other words, sometimes the procedural parts of a program are not even necessary to fully convey the programmer's algorithm. VHDL's types are not nearly as powerful, but they still offer a high degree of clarity on what the semantics of a program should look like given the types of its inputs and outputs.

The Mechanics of Type Systems

Type systems have many uses ranging from optimization to documentation, but one of the primary motivations behind type systems is to determine if programs will cause errors when they are being run (runtime). The obvious way to check if a program will cause an error is to run the program, but this process can be messy. Programs can take a long time to run and in the case of hardware description languages, running programs requires either simulating them using complicated software or flashing them to hardware. If a program *can* generate a runtime exception then it certainly not guaranteed to do so. This means that to guarantee that a program will not generate a runtime error, every possible execution path through the program must be tested. For very large, interconnected

systems, this is not practical.

A type system solves this problem by analyzing the text of a program without running it. The type checker avoids the issue of convergence and can run far more quickly than the program under analysis. But what does the type checker analyze? The type checker approximates the *values* of the program with *types* [4].

There are some aspects of a value that cannot be captured by its type. But, type systems have been shown to be massively effective at detecting errors in programs despite the fact that they cannot detect every error that might occur at runtime.

Typing Relations

A type system is specified by inference rules on the terms of the language. Terms are the syntactic elements that make up a program at the text level. They are also easily represented by a tree where terms may have one or more children. Figure 4 demonstrates inference rules for checking expressions in a language where the only values are booleans and the only terms are if expressions. In this example, `true`, `false`, and `if` t_1 `then` t_2 `else` t_3 are terms, and `if` has three children: t_1, t_2, and t_3 which are allowed to be any other term in the language. Programs in this simple language are trees made up of these three components. In a more complex language, there may be tens of possible terms.

The typing relation is composed of inference rules. Each inference rule can be seen as a mapping from terms to types. Each inference rule has a conclusion (below the line) and a list of conditions (above the line). The statement below the line holds if all of the conditions above the line are true.

The rules in Figure 4 show how to assign a term in the language to a type. The T-True and T-False rules have no conditions. This is because when `true` or `false` appear in the language, they *must* have type Bool. The T-If rule is more complicated. The type of the if expression depends on the types of terms t_2 and t_3.

$$\boxed{t : T}$$

$$\frac{}{\texttt{true} : \text{Bool}}\ \text{T-True}$$

$$\frac{}{\texttt{false} : \text{Bool}}\ \text{T-False}$$

$$\frac{t_1 : \text{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T}\ \text{T-If}$$

Figure 4: Typing rules for booleans

The inference rules also show which compositions of terms in the language are valid. For example, the T-If inference rule specifies the condition that t_1 has type Bool. If t_1 is evaluated to have a type other than bool, this constitutes an invalid program. Using the same type variable, T, to describe the type of t_2 and t_3, the T-If rule also specifies that the two branches of the if must have the same type.

The inference rules also give us the outline of an algorithm that can be used to check the type of a term. To determine the type of a composite term, the rules imply that we must *recurse* over the terms, finding the type of each subterm and using it to inform the type of the current term.

Therefore, the type system can be seen as performing an automatic proof of the validity of the terms by *induction* over the terms. Terms that have trivial conditions such as T-True and T-False are considered base cases, and terms that contain subtrees are inductive cases [4].

The type system of VHDL is somewhat more complicated than the example provided, but its type system can be seen as an extension of the system presented.

Looking Forward

Though VHDL and Verilog remain very prominent in the landscape of HDLs, new languages have appeared and are gaining traction. One interesting class of languages that are now being used for hardware description are functional programming languages. Functional languages such as OCaml and Haskell are often accompanied by complex, versatile type systems, making it possible to write libraries that function as domain-specific languages whose compiler is implemented in the host language itself. Further, functional languages

make use of polymorphism, higher-order functions, and other language features that transfer well to hardware description languages, but that aren't present in VHDL or Verilog. Though simulation and validation of hardware is certainly possible inside functional HDL libraries, most libraries convert the designs to VHDL or Verilog for export to the manufacturers toolchain for synthesis to actual FPGA hardware.

Conclusion

Type systems are an excellent tool for validation of programs and documentation of a programmer's intent. VHDL and Verilog use strong type systems both to make guarantees about the correctness of programs and to provide embedded documentation about the hardware designs that they describe. In an environment where data is entirely homogeneous without the use of types, type systems provide an important information carrying layer in hardware description languages, a role which is less important in traditional programming languages.

References

[1] David R. Coelho (1989). *The VHDL Handbook*. Springer Science & Business Media. ISBN 978-0-7923-9031-2

[2] Iván Bach. 1982. On the type concept of Ada. Ada Lett. II, 3

[3] Polikarpova, N. Kuraj, I., & Solar-Lazama, A. (2016). Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices, 51(6)*, 522-538

[4] Pierce, B. C. (2002). *Types and programming languages*.