Compilers Research

Aditya Srinivasan, Drew Hilton

August 2017

1 Preface

This resource aims to document and formalize aspects of the compilers research project that I am undertaking. It will be updated regularly as the project evolves so as to reflect the most recent and relevant decisions in order to serve as a reference.

2 Overview

This research project is an attempt to develop a programming language for describing hardware and an accompanying compiler. Currently, hardware description languages lack features that would improve ease of expression and modularity, and reduce the potential for errors during execution. Such features include, but are not limited to, strong type systems, parametric polymorphism, and user-defined datatypes. Although there exist languages that strive to provide these primitives, they do not fully consider the semantics of hardware design. The compiler developed through this research project will use the newly developed language as the source and compile to Verilog as the target.

3 Formalizing the Dichotomy

As mentioned in Section 2, there is a need to formalize the dichotomy between the software and hardware components so as to develop a language that respects both entities appropriately. As such, in this section, we will distinguish the types, semantics, and syntax of the two.

3.1 Types

We will first formalize the definitions of the two categories of types: software (SW) and hardware (HW).

```
hw = bit
   | hw[]
   | (hw * hw)
```

(a) HW types

```
sw = int
   | float
   | hw
   | sw list
   | (sw * sw)
   | sw -> sw
```

(b) SW types

Figure 1: Type definitions

As can be seen in Figure 1(a) above, the hardware type is defined as a bit, a fixed-sized array of a hardware type, or a tuple of hardware types. In order to justify this, we argue that all data is represented by bits in hardware, and as such the fundamental data representation of a hardware type must be a bit. We augment the expressive power by introducing fixed-sized arrays, in order to represent bit vectors or bit patterns. Finally, we permit the declaration of tuples in order to group hardware types into one value.

Note that the type declarations for the fixed-size array and tuple are recursive, in that any hardware type can populate these types. This allows for the hardware expression of higher-dimensional arrays, arrays of tuples of bits, tuples of arrays of bits, et cetera.

In Figure 1(b), the software type is defined as an int, a float, a hardware type, a dynamically-sized list of software types, a software tuple, or a software-to-software function. The primitive types we provide in this declaration are integers, floating point numbers, dynamically-sized lists, tuples, and functions. The software type is again defined recursively, allowing more complex types to be expressed. We also note that the set of software types is a superset of the set of hardware types. This is so that any hardware component can be described and represented in the software, which is essential in designing a hardware description language.

3.2 Declarations

Given that the hardware and software types are distinct, we must formalize the syntax to declare such types. Later, we will discuss the mechanisms by which to perform explicit conversions, as there are is implicit data conversion.

3.2.1 Hardware

A bit is declared using an apostrophe, the letter 'b', followed by the bit value:

```
val a := 'b0
```

A fixed-size array of hardware types can be declared in multiple ways, each of which must explicitly state the size of the array.

In the most general case, an array can be declared using the following syntax:

```
val b := array [32] of 'b0
```

In the above declaration, the type of the variable `b` is `bit[32]`. In other words, the size of the array parameterizes the type. With the concept of generics, which will be discussed later, functions may strictly permit certain sized arrays, or accept arbitrarily sized arrays. In general the type of an array of size `n` for some hardware type `` `a{hw} `` is `` `a{hw}[n] ``.

Some other ways of declaring arrays are shown below:

```
val c := {'b0, 'b1, 'b0, 'b0}
val d := 32'b10011001
```

The parameter (i.e. size) of the type of variable `c` is never explicitly stated, but can be inferred from the literal declaration. The size parameter of the type of variable `d` is stated, and the initial value provided occupies left-most elements of the array, with the remainder being padded by 0s. In other words, the following declarations are equivalent:

```
val d1 := 8'b1011
val d2 := {'b1, 'b1, 'b0, 'b1, 'b0, 'b0, 'b0, 'b0}
```

The final syntactical specification for the hardware types is for the tuple. These represent pairs or groups of other hardware values, and are declared in one way as follows:

```
val e := ('b0, array [32] of 'b0)
```

In the above declaration, the type of the variable `e` is `(bit * bit[32])`. In general the type of a tuple with hardware types `` `a{hw} `` and `` `b{hw} `` is expressed as `` (`a{hw} * `b{hw}) ``.

This concludes the syntax for declaring hardware types, and their representation. Since the definitions are recursive, one can define arbitrarily complex types if desired while still forming a type that is expressible in hardware.

3.2.2 Software

One of the software types is the hardware type. This means that any of the aforementioned types can be represented in software which is obviously necessary in order to write programs in this language.

The first pure software type is the integer. Although integers are represented as arrays of bits in hardware, they are not explicitly interpreted as such. The semantics of an integer do not exist in the realm of hardware, and as such this type is defined solely in software. Integers adhere to the same semantics as in most conventional programming languages. The syntax for declaring an integer is as follows:

```
val f := 1024
val g := -42
```

Another pure software type is the floating point decimal. These are also represented as sequences of bits in hardware, although that is not always their interpretation. Floats adhere to the same semantics as in most conventional programming languages, with the following syntax:

```
val h := 5601.23
val i := -101.
```

Our first augmentation of the software type introduces the dynamically-sized list. There is no strict size specification for such entities, as they can contain software types and do not need to be translated directly to hardware. The syntax to declare a list is as follows:

```
val j := [2, 3, 5, 7, 11, 13, 17]
val k := 2::[3, 5, 7, 11, 13, 17]
```

The first expression is a literal declaration of a list of the first 7 prime numbers. The second expression results in the same list, but leverages the `::` operator in order to append an element to the list. This operator, and more, will be discussed in a later section.

Similarly to the tuples seen in section 3.2.1, there exist software tuples, which are declared in the same syntax:

```
val l := (2, 10.5)
val m := ([1, 2, 3], 'b0)
```

Although tuples are declared in the same way for software and hardware types, the compiler can perform type inference to determine the kind of tuple. A tuple composed purely of hardware types is a hardware tuple, whereas a tuple containing any software type is a software tuple. We stray from the fact that a hardware type is a software type in this case when determining the type category of values.

The last software type is the function. This is an extremely powerful construct that allows some function to be performed on a software-typed input to produce a software-typed output. The general declaration for a function is as follows:

```
fun f arg0 arg1 . . . argn = res
```

If `arg0` has type `'a0`, `arg1` has type `'a1`, and so on, and `res` has type `'r`, then the function `f` has type `'a0 -> 'a1 -> . . . -> 'an -> 'r`.

Since functions themselves are values, a function may be passed as a parameter to a different function. This is demonstrated by the `map` function, which has signature `('a -> 'b) -> 'a list -> 'b list`. Similarly, tuples can be passed as values, as demonstrated by the `#1` function, which has the signature `('a * . . . ) -> 'a`.

3.3 Operations

With a specification of the syntax responsible for declaring various types, we must now discuss the syntax for performing operations on these. Given our language is strongly typed, it does not support operator overloading as type inference would then fail. Instead, there are distinct operators that only permit certain types to be provided.

3.3.1 Bitwise operators

The first kind of operator we introduce is the bitwise operator. These operators accept any hardware type. It should be noted that the binary operators in this class must be supplied hardware types of the same exact structure. That is, an `and` operation cannot be performed between a tuple of bits and an array of bits.

1. `&`: performs a bitwise and between corresponding bits
2. `|`: performs a bitwise or between corresponding bits
3. `^`: performs a bitwise xor between corresponding bits
4. `~`: performs a bitwise complementation of bits

The first three operators in the above list are binary and (in prefix form) have the functional type `'a -> 'a -> 'a`. The fourth operator in the above list is unary and has the functional type `'a -> 'a`.

3.3.2 Arithmetic operators

The second kind of operator we introduce is the arithmetic operator. These operators only accept the software constructs integer and float. Furthermore, as mentioned earlier, there is no operator overloading so arithmetic between integers and floats differ in syntax.

The operators for integer arithmetic are shown below.

1. `+`: integer addition
2. `-`: integer subtraction
3. `/`: integer division
4. `*`: integer multiplication
5. `%`: integer modulo

All of these operators (in prefix form) have the type `int -> int -> int`.

The operators for floating point arithmetic are shown below

1. `+.`: floating point addition
2. `-.`: floating point subtraction
3. `/.`: floating point division
4. `*.`: floating point multiplication

All of these operators (in prefix form) have the type `float -> float -> float`.