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 Language Specification

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 informally distinguish the types, semantics, and syntax of the two. The formalizations of the dichotomy, including proofs regarding certain desirable properties of the language and semantics, will be offered in Section 4.

3.1 Types

We will first distinguish between three categories of types, or kinds: software values , hardware data, and hardware modules. A more formal description of these kinds is offered in Section 4.2, while we limit ourselves to a BNF diagram here to illustrate which types belong to which kind.

$\langle S \rangle$ ::= `int`
| `real`
| $\langle H \rangle$
| $\langle M \rangle$
| $\langle S \rangle$ `list`
| `{a:` $\langle S \rangle$ `, b:` $\langle S \rangle$ `,...}`
| `(` $\langle S \rangle$ `*` $\langle S \rangle$ `)`
| $\langle S \rangle$ `ref`
| $\langle S \rangle \rightarrow \langle S \rangle$
| `unit`

add 'sw' constructors

T_S notation

$\langle H \rangle$ ::= `bit`
| $\langle H \rangle$`[int]`
| `#(`$\langle H \rangle$ `*` $\langle H \rangle$ `)`
| $\langle H \rangle$ `@ int`

$\langle M \rangle$ ::= $\langle H \rangle \rightarrow \langle H \rangle$

Figure 1: Kind definitions

The set of types in kind H has the property that all members represent data that can be manifest as values in hardware. Specifically, this kind contains the following types: a bit, a fixed-sized array of a type in kind H, a tuple of types in kind H, or a timed value of a type in kind H. 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 value type must be a bit. We augment the expressive power by introducing fixed-sized arrays, in order to represent bit vectors. Next, we permit the declaration of tuples in order to group multiple hardware data values into one value. Finally, we introduce timed hardware values since latched data, or pipelined data, is subject to certain temporal constraints.

Note that the kind declaration is recursive in the definition of the fixed-size array, tuple, and temporal type, in that any type belonging to H can populate these types. This allows for the expression of higher-dimensional arrays, arrays of tuples of bits, tuples of arrays of bits, et cetera ad infinitum.

The kind M governs all functions that accept a type of kind H and output a type of kind H. These "hardware functions", named formally as "modules", are designed to prohibit the existence of higher-order modules, which do not comply with the semantics of hardware design. More explicitly, since it is not possible for anything other than a type of kind H to be the input or output of a module, the module is well-defined in hardware.

In Figure 1, the kind S is defined by the type elements `int`, `real`, a type of kind H or M (either hardware data values or hardware modules), a dynamically-sized `list` of types in kind S, a record of named fields each of a type in kind S, a tuple of types in kind S, a reference (`ref`) of a type in kind S, a function from a type in kind S to a type in kind S, or the type `unit`.

The software kind is again defined recursively, allowing more complex types to be expressed. We also note that the set of elements in S is a superset of the set elements in both H and M. This is so that any hardware value or module can be described and represented in the software, which is essential in designing a hardware description language such that programmers can create and manipulate such elements.

3.2 Declarations

Given that the hardware and software types are distinct, we must formalize the syntax to declare such types. In a later section we will discuss the mechanisms by which to perform explicit conversions, as there is no implicit data conversion.

3.2.1 Hardware

A bit is declared in literal syntax using an apostrophe, the sequence 'b:', followed by the bit value:

```
'b:0
```

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

To declare an array literal, one can use the following syntax:

```
#['b:0, 'b:1, 'b:0, 'b:0]
```

In the above declaration, the type of the variable `b` is `bit[4]`. 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 value type `'hd` is `'hd[n]`.

An array may also be declared in index-functional form. In this form, the programmer specifies the size of the array and a function mapping from the bit index to a value. For example:

```
#[32, i -> if(i % 3 = 0) then 'b:1 else 'b:0]
#[32, 0 -> 'b:0 | i -> d[i - 1] & c[i]]
```

Here we employ a special syntax for declaring the index-to-value function that allows the expression of pattern-matching succinctly. This differs from conventional pattern-matching syntax, which will be discussed later.

The ability to specify bit patterns as representations of integers or floats is also provided with the following syntax:

```
32's:-42
64'u:x
(40, 7)'f:10.499
```

The above syntax is shorthand for the explicit conversion functions covered in section 3.4.1, but are featured for succinctness. The first declaration creates a bit array of size 32 representing the signed integer 42. The second creates a bit array of size 64 representing the unsigned integer contained in variable `x`. The third creates a bit array of size 48 representing the floating point decimal 10.499 with 1 (implicitly defined) sign bit, 7 exponent bits and 40 mantissa bits.

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:

```
#('b:0, #['b:0, 'b:1, 'b:1, 'b:0])
```

In the above declaration, the type of the variable `e` is `(bit * bit[4])`. In general the type of a tuple with hardware types `` `hd1 `` and `` `hd2 `` is expressed as `` \#(`hd1 * `hd2) ``.

Finally, one may express a hardware value with temporal type using the following syntax:

```
'b0 with {latency = 4}
```

In the above declaration, the type of the variable `f` is `bit @ 4`. The above declaration leverages the annotation syntax, which will be elaborated upon in a later section.

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

The hardware module has the following general syntax:

```
module j x := ...; y
```

There are a few things to note. Firstly, `g` is declared as a `module` instead of a `val` or `fun`. Secondly, it is supplied a single argument, in this case named `x`. Lastly, after performing any number of operations, it returns a single value, in this case named `y`. The type of any module is `'hd1 → 'hd2` where `'hd1` and `'hd2` are arbitrary hardware data value types belonging to kind H.

3.2.2 Software

All types belonging to H and M also belong to S. This means that any of the aforementioned types can be represented in software.

The first purely 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, although methods to express integers (both signed and unsigned) as bit vectors exist. Integers adhere to the same semantics as in most conventional programming languages. The syntax for declaring an integer is as follows:

```
1024
-42
#'b:1100101
#'o:72034
#'x:deadb33f
```

Another pure software type is the real, or floating point decimal. These are also represented as bit vectors in hardware, although that is not always their interpretation. Reals adhere to the same semantics as in most conventional programming languages, with the following syntaxes:

```
5601.23
-101.1
.382
+192.
-1003.47e+07
423E-7
```

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 any type of kind S and do not need to be translated directly to hardware. The syntax to declare a list is as follows:

```
[2, 3, 5, 7, 11, 13, 17]
2::[3, 5, 7, 11, 13, 17]
```

Note that it is possible to declare a dynamically-sized list of hardware value types, since the notation for a fixed-sized array includes the pound sign, but

such a list will not be expressible as a hardware value. In the above syntax, 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 hardware tuples seen in section 3.2.1, there exist software tuples, which are declared with a similar syntax:

```
(2, 10.5)
([1, 2, 3], 'b:0)
```

The record and tuple types are similar in that values are grouped together. In fact, as will be elaborated upon further, tuples are merely syntactic sugar for records. In a record, fields are named in a manner specified by the programmer. Tuples are records in which the fields are the natural numbers, beginning from `1` and incrementing for however many elements are in the tuple. The syntax for declaring records is as follows:

```
{a = 1, b = 2.5, c = (x > y, [1, 2, 3])}
```

In the above declaration, the record has type `\{a: int, b: float, c: (int * int list)\}`.
The unit type is merely an alias for the empty tuple type. The syntax or declaring a unit value is as follows:

```
()
```

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

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

If `arg0` has type `'sw0`, `arg1` has type `'sw1`, and so on, and `res` has type `'swr`, then the function `f` has type `'sw0 → 'sw1 → . . . → 'swn → 'swr`.

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. These operators can be thought of as functions, and the operands can be considered the arguments.

3.3.1 Bitwise operators

The first kind of operator we introduce is the bitwise operator. These operators accept any hardware value type with the restriction that any two types supplied to these operators must be exactly the same. That is, a bitwise-and operation cannot be performed between a tuple of bits and an array of bits.

1. `&`: bitwise-and
2. `|`: bitwise-or
3. `^`: bitwise-xor
4. `~`: bitwise-complementation
5. `<<`: logical left shift
6. `>>`: logical right shift
7. `>>>`: arithmetic right shift

The first three operators in the above list are binary and (in prefix form) have the functional type `'hd → 'hd → 'hd`. The fourth operator in the above list is unary and has the functional type `'hd → 'hd`. The last three operators are binary and have the functional type `bit[n] → bit[m] → bit[n]`. The first argument is the bit pattern to shift, the second argument is the bit pattern representing the amount to shift, and the result is the shifted bit pattern. Often, calls to these shift operators will require conversions from integers to bit patterns, which are discussed in section 3.4.1.

3.3.2 Arithmetic operators

The second kind of operator we introduce is the arithmetic operator. These operators only accept integer and real types. Furthermore, as mentioned earlier, there is no operator overloading so arithmetic between integers and reals 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`. These operators operate as if the integers provided are signed. If explicit unsigned integer operations are desired, built-in structures can be used, which will be elaborated in a later section.

The operators for floating point arithmetic are shown below

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

All of these operators (in prefix form) have the type `real` → `real` → `real`.

3.3.3 Conditional operators

In this language, integers possess boolean semantic value. An integer value of 0 denotes a false condition, whereas any other integer value denotes a true condition. Using this system, we define three conditional operators.

1. `andalso`: logical conjunction
2. `orelse`: logical disjunction
3. `not`: logical negation

The first two operators are binary and (in prefix form) have functional type `int` → `int` → `int`. The last operator is unary and has type `int` → `int`.

3.3.4 Comparison operators

The language also provides operators to perform comparisons of various types. These operators are generic, and when declaring types they may be overridden to specify the semantics of comparisons for that type, which will be discussed in a later section. For now, the comparison operators are as follows:

1. `=`: equal
2. `<>`: not equal
3. `>`: greater-than
4. `>=`: greater-than-or-equal-to
5. `<`: less-than

6. `<=`: less-than-or-equal-to

All of these operators (in prefix form) have type `'a → 'a → int`, where the returned integer represents the boolean result, namely 0 indicating falsehood and 1 indicating truth.

The core library provides definitions of these comparison operators for bits, bit vectors, integers, and reals.

3.3.5 List operators

As mentioned earlier, there is a single list operator which allows for the appending of an element to the beginning of a list.

1. `::`: element concatenation `'a → 'a list → 'a list`

3.3.6 Tuple operators

The language provides built-in operators – which, recall, are no different from functions – intended for use with tuples.

1. `#i`: element retrieval `('a1 * ... * 'ai * ... * 'an) → 'ai`

3.3.7 Record operators

As with tuples, there exists an operator to retrieve the field from a record given its name.

1. `#i`: field retrieval `a: 'a, b: 'b, ..., z: 'z → 'i`

3.3.8 Reference operators

References have an important operator that allows the underlying value to be retrieved, via a process known as "dereferencing".

1. `!`: dereferencing `'a ref → 'a`

3.4 Conversions

With the above established syntax for declaring and operating on types, there is a need to further establish a system by which one can convert from a given type to another. For example, when performing a shift, programmers must specify the shift amount as a bit pattern. It may be desired to shift according to the value of some integer, in which case a conversion must be made explicitly before calling the operator. This further enforces the dichotomy by making programmers explicitly cast from one type to another.

3.4.1 Software-to-hardware

The functions by which to convert software values to hardware values are namespaced by the structure corresponding to the value from which the conversion is being applied.

The function `toBit` of the structure `Int` converts an integer to a bit value. Namely, `0` converts to `'b:0` and any other integer converts to `'b:1`. Since integers are used to represent boolean condition values, this mechanism helps in converting a boolean condition into a bit value.
The function `unsignedToBits<n>` of the structure `Int` converts a integer to a bit array of size `n` by interpreting the integer as unsigned. This function has type `int` $\rightarrow$ `bit[n]`, and returns the a sequence of bits corresponding to the input integer, treated in unsigned form. Note that the output bit array can be of any size, as its size type is parameterized by the function call. The supplied integer may not be expressible in the provided number of bits, in which case an error will be raised. The function `signedToBits<n>` of the structure `Int` acts similarly, although the produced bit array corresponds to a signed representation of the input number.

The function `toBits<s, e, m>` of the structure `Real` converts a floating point value to a bit array according to the convention supplied by parameters `s`, `e`, and `m`. In particular, these represent the number of sign bits, exponent bits, and mantissa bits respectively, used to assemble the output bit array. This function has type `real -> bit[n]` where `s + e + m` $\equiv$ `n`, and returns the bit array representing the input floating point value.

Finally, the function `toArray` of the structure `List` converts a software list to a fixed size array according to a supplied mapping function. The function has signature `('a -> 'hd) -> 'a list -> 'hd[n]`, where the first argument is the function, the second argument is the software list, and the result is the hardware fixed-size array.

3.4.2 Hardware-to-software

Since hardware values represent actual physical wires and circuits, it isn't possible to always retrieve and store their value in the typical semantics of a software variable. Therefore, the ability to convert from hardware to software is limited to a surface-level conversion from a hardware representation of a bit to a software representation of a bit. This is done using the function `fromHW` in the `Bit` structure.

3.5 Temporal typing

The semantics of temporal typing are discussed in this section. As an example, consider the built-in function `dff` which has type `'a` $\rightarrow$ `'a @ 1`. This indicates

that the output of the function is only valid after 1 clock cycle.

In the case that temporal types are not explicitly defined, the compiler infers the temporal type of a function. For example, consider the following function:

```
fun f a b c =
    let
        val x = dff(dff(a))
        val y = dff(dff(b))
        val z = dff(dff(c))
    in
        #(x, y, z)
    end
```

This function has the type `('a * 'b * 'c) → ('a @ 2 * 'b @ 2 * 'c @ 2)`. This is because all outputs have the same clock delay.

In the event where there are differing clock delays, we specify the temporal type for each output. For example, consider the modified function:

```
fun f a b c =
    let
        val x = dff(a)
        val y = dff(dff(b))
        val z = dff(dff(dff(c)))
    in
        #(x, y, z)
    end
```

This function has the type `('a * 'b * 'c) → ('a @ 1 * 'b @ 2 * 'c @ 3)`.
In the event where outputs are formed as a result of different temporal types, we state that the temporal type is undefined. For example, consider the further modified function:

```
fun f a b c =
    let
        val x = dff(a)
        val y = dff(dff(b))
        val z = dff(dff(dff(c)))
    in
        #(x > y, y = z, z)
    end
```

Now, function `f` has type `('a * 'b * 'c) → ('a @ ?  * 'b @ ?  * 'c @ 3)`.

Functions can enforce the temporal type of parameters so as to prevent timing errors.

3.5.1 Temporal lengthening

In the event that a function expects an argument of type `'a @ n`, but the programmer has an argument of type `'a @ m` where $m < n$, there exists a built-in function to adjust the temporal type. Namely, `lengthen(x, n)` accepts a hardware value and extends its temporal type to `n`, giving it the type `'hv @ m` $\rightarrow$ `'hv @ n`. In order for this to succeed, it must hold that $m \leq n$.

```
/* suppose f has type ('hv @ 2 -> 'hv @ 3) */
val a = dff('b:0)
val b = f a    /* fails type-checking */
val c = f lengthen(a, 2)
```

Two hardware values can also by synchronized to have the same temporal type using the function `sync` which has type `('hd1 @ n * 'hd2 @ m) -> ('hv1 @ t * 'hv2 @ t)` where `t` $\equiv$ `max(n, m)`.

3.6 Elements

Previous subsections discussed certain syntax and design decisions in order to strengthen the dichotomy between the software and hardware entities. In this subsection, we discuss some constructs of the programming language that are generally useful.

3.6.1 Let-bindings

Let-bindings are common in functional programming languages, and allow for the programmer to bind variables to values within a certain scope, before and after which the binding does not exist.
The syntax for a let-binding is as follows:

```
let
    /* value, function, type,
    datatype, module declarations */
in
    /* expression */
end
```

The resulting value of a let-binding is the result of the expression between the `in` and `end` keywords.

3.6.2 Pervasives

The programming language allows programmers to name pervasive elements, such as clock or reset wires, that can be accessed across modules without explicit parameter passing.

The programmer defines such elements using the `pervasive` keyword, defined outside of the scope of any module. These pervasive elements exist in the scope of all specified modules, and can be accessed naturally. For example:

```
pervasive clk : bit in moduleA, moduleB, moduleC;
module moduleA x = dff(x, clk)
module moduleB y = (moduleA y) ^ dff(y, ~clk)
module moduleC (z, opcode) = alu(z, opcode, clk)
```

The type of the pervasive element must be declared in order to provide strong type-checking for all modules that use the element.

3.6.3 Types and Datatypes

Programmers may define custom types and datatypes using primitives or other types they have defined. The syntax for declaring a type is as follows:

```
type my_type = bit * bit[32]
```

The ability to override comparison operators is provided, in case certain semantics are desired:

```
type my_type = bit * bit[32]
with operator > (a, b) = (#1 a) > (#1 b)
```

The syntax for declaring a datatype is as follows:

```
datatype my_datatype = FOO of bit * bit[32]
                     | BAR of bit * bit[64]
```

These are useful in conjunction with case statements to do pattern-matching.

3.6.4 Structures and Signatures

Software structures are constructs that can contain value, type, datatype, function, and module definitions. There are many built-in structures such as Int, Real, Bit, List, Set, Map, and Queue. The syntax for declaring a structure is as follows:

```
structure MyStruct =
struct
    /*
     * value, type, datatype, function, or module declarations
     */
end
```

If desired, structures can be made to adhere to signatures, which act as interfaces. There are two ways to declare signatures. The first is using a separate signature declaration:

```
signature MySig =
sig
    /*
     * value, type, datatype, function, or module definitions
     */
end

structure MyStruct : MySig =
struct
    /*
     * value, type, datatype, function, or module declarations
     */
end
```

The second way is by declaring the signature in-line with the structure declaration:

```
structure MyStruct :
    sig
        /*
         * value, type, datatype, function, or module definitions
         */
    end =
struct
    /*
     * value, type, datatype, function, or module declarations
     */
end
```

4 Formalizations

In the previous section we established, informally, various specifications of the language. We will now draw attention to various formalizations of the program-

ming language.

4.1 Grammar and Rules

First, we develop a set of grammars and rules that define the programming language. The semantic style we will be assuming is that of operational semantics. In the next sections, we will develop the grammars for values and types, as well as the rules for evaluation and typing.

4.1.1 Value Grammar

The first grammar defines the possible values, which are final results of evaluation that cannot be reduced any further. We divide our grammar definition into two sections: one to define the grammar of software values and another for hardware values.

⟨*sw-value*⟩ ::= ⟨*integer*⟩
| ⟨*real*⟩
| ⟨*string*⟩
| ⟨*list*⟩
| ⟨*record*⟩
| ⟨*sw-tuple*⟩
| ⟨*function*⟩
| ⟨*ref*⟩

⟨*integer*⟩ ::= ⟨*binary-integer*⟩
| ⟨*octal-integer*⟩
| ⟨*decimal-integer*⟩
| ⟨*hex-integer*⟩

⟨*binary-integer*⟩ ::= `#'b:` ⟨*binary-digits*⟩

⟨*octal-integer*⟩ ::= `#'o:` ⟨*octal-digits*⟩

⟨*decimal-integer*⟩ ::= ⟨*decimal-digits*⟩
| ⟨*sign*⟩ ⟨*deciaml-digits*⟩

⟨*hex-integer*⟩ ::= `#'x:` ⟨*hex-digits*⟩

⟨*integer-tail*⟩ ::= ⟨*integer-tail*⟩ ⟨*decimal-digit*⟩
| ⟨*decimal-digit*⟩

⟨*real*⟩ ::= ⟨*real-tail*⟩
| ⟨*sign*⟩ ⟨*real-tail*⟩

⟨*real-tail*⟩ ::= ⟨*decimal-digits*⟩ `.` ⟨*decimal-digits-or-empty*⟩ ⟨*exponent-or-empty*⟩
| ⟨*decimal-digits-or-empty*⟩ `.` ⟨*decimal-digits*⟩ ⟨*exponent-or-empty*⟩
| ⟨*decimal-digits*⟩ ⟨*exponent*⟩

⟨*decimal-digits-or-empty*⟩ ::= ⟨*decimal-digits*⟩
| ϵ

⟨*exponent*⟩ ::= `E` ⟨*decimal-digits*⟩
| `E` ⟨*sign*⟩ ⟨*decimal-digits*⟩
| `e` ⟨*decimal-digits*⟩
| `e` ⟨*sign*⟩ ⟨*decimal-digits*⟩

⟨*exponent-or-empty*⟩ ::= ⟨*exponent*⟩
| ϵ

⟨*binary-digit*⟩ ::= `any of 0 or 1`

⟨*octal-digit*⟩ ::= `any of 0, 1, 2, 3, 4, 5, 6, or 7`

⟨*decimal-digit*⟩ ::= `any of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9`

⟨*hex-digit*⟩ ::= `any of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, or F`

⟨*binary-digits*⟩ ::= ⟨*binary-digits*⟩ ⟨*binary-digit*⟩
| ⟨*binary-digit*⟩

⟨*octal-digits*⟩ ::= ⟨*octal-digits*⟩ ⟨*octal-digit*⟩
| ⟨*octal-digit*⟩

⟨*decimal-digits*⟩ ::= ⟨*decimal-digits*⟩ ⟨*decimal-digit*⟩
| ⟨*decimal-digit*⟩

⟨*hex-digits*⟩ ::= ⟨*hex-digits*⟩ ⟨*hex-digit*⟩
| ⟨*hex-digit*⟩

⟨*sign*⟩ ::= `+`
| `-`

⟨*string*⟩ ::= `"` ⟨*string-tail*⟩ `"`

⟨*string-tail*⟩ ::= ⟨*string-tail*⟩ ⟨*string-character*⟩
| ϵ

⟨*string-character*⟩ ::= `any printable character, including space, except for " and \`
| `\` ⟨*escape-character*⟩

⟨*escape-character*⟩ ::= `\`
| `'`
| `"`
| `a`
| `b`
| `e`
| `f`
| `n`
| `r`
| `t`
| `0`
| ⟨*hex-digits*⟩

⟨*list*⟩ ::= `[` ⟨*list-body*⟩ `]`
| ⟨*sw-value*⟩ `::` ⟨*list*⟩
| `nil`

⟨*list-body*⟩ ::= ⟨*sw-value*⟩ ⟨*list-body*⟩
| `,` ⟨*list-body*⟩
| ϵ

⟨*record*⟩ ::= { ⟨*record-body*⟩ }

⟨*record-body*⟩ ::= ⟨*identifier*⟩ `=` ⟨*sw-value*⟩ ⟨*record-body*⟩
| `,` ⟨*record-body*⟩
| ϵ

⟨*function*⟩ ::= $\lambda x.e$ T_S.<sw-term>

⟨*ref*⟩ ::= `ref` ⟨*sw-value*⟩

The following defines the grammar for hardware values.

⟨*hw-value*⟩ ::= ⟨*hw-data*⟩
| ⟨*hw-module*⟩

⟨*hw-data*⟩ ::= ⟨*bit*⟩
| ⟨*array*⟩
| ⟨*hw-tuple*⟩
| ⟨*temporal-value*⟩

⟨*bit*⟩ ::= `'b:` ⟨*binary-digit*⟩

⟨*array*⟩ ::= `#[` ⟨*array-body*⟩ `]`

⟨*array-body*⟩ ::= ⟨*hw-value*⟩ ⟨*array-body*⟩
| `,` ⟨*array-body*⟩
| ϵ

⟨*hw-tuple*⟩ ::= `#(` ⟨*hw-tuple-body*⟩ `)`

⟨*hw-tuple-body*⟩ ::= ⟨*hw-value*⟩ ⟨*hw-tuple-body*⟩
| `,` ⟨*hw-tuple-body*⟩
| ϵ

⟨*temporal-value*⟩ ::= ⟨*hw-value*⟩ `@` ⟨*integer*⟩

⟨*hw-module*⟩ ::= $\lambda x.e$ T_H.<hw-term>

4.1.2 Term Grammar

The next grammar defined is that of terms. These are expressions that are not final results of evaluation, but which result in some value after an arbitrary number of evaluation steps.

⟨*sw-term*⟩ ::= ⟨*identifier*⟩
| ⟨*conditional*⟩
| ⟨*operations*⟩
| ⟨*assign*⟩
| ⟨*bit-array*⟩

tick-id as a token for bit-array conversions
exp TID COLON exp

⟨*identifier*⟩ ::= ⟨*id-start*⟩ ⟨*id-tail*⟩

⟨*id-start*⟩ ::= `any alphabetic character or underscore`

⟨*id-tail*⟩ ::= ⟨*id-tail*⟩ `any alphanumeric character or underscore`
| ϵ

⟨*conditional*⟩ ::= `if` ⟨*sw-term*⟩ `then` ⟨*sw-term*⟩ `else` ⟨*sw-term*⟩
| `if` ⟨*sw-term*⟩ `then` ⟨*sw-term*⟩

⟨*operations*⟩ ::= ⟨*arith-op*⟩
| ⟨*bit-op*⟩
| ⟨*compare-op*⟩
| ⟨*list-op*⟩

⟨*arith-op*⟩ ::= ⟨*int-op*⟩
| ⟨*real-op*⟩

⟨*int-op*⟩ ::= `-` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `+` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `-` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `*` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `/` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `%` ⟨*sw-term*⟩

⟨*real-op*⟩ ::= ⟨*sw-term*⟩ `+.` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `-.` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `*.` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `/.` ⟨*sw-term*⟩

⟨*bit-op*⟩ ::= `~` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `|` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `&` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `^` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `<<` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `>>` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `>>>` ⟨*sw-term*⟩

⟨*compare-op*⟩ ::= ⟨*sw-term*⟩ `=` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `<>` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `>` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `<` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `>=` ⟨*sw-term*⟩
| ⟨*sw-term*⟩ `<=` ⟨*sw-term*⟩

⟨*list-op*⟩ ::= ⟨*sw-term*⟩ `::` ⟨*sw-term*⟩

⟨*assign*⟩ ::= ⟨*sw-term*⟩ `:=` ⟨*sw-term*⟩

⟨*bit-array*⟩ ::= ⟨*decimal-digits*⟩ `'u:` ⟨*decimal-digits*⟩
| ⟨*decimal-digits*⟩ `'s:` ⟨*decimal-integer*⟩
| `(` ⟨*decimal-digits*⟩ `,` ⟨*decimal-digits*⟩ `)'r:` ⟨*real*⟩

Below is the grammar for derived terms.

⟨*term-sequence*⟩ ::= ⟨*term-sequence*⟩ `;` ⟨*sw-term*⟩
| ⟨*sw-term*⟩

⟨*sw-tuple*⟩ ::= `(` ⟨*sw-tuple-body*⟩ `)`

⟨*sw-tuple-body*⟩ ::= ⟨*sw-value*⟩ ⟨*sw-tuple-body*⟩
| `,` ⟨*sw-tuple-body*⟩
| ϵ

4.2 Types and Kinds

We now formalize the kinding system of the language. As mentioned in section 3.1, there are three kinds: S, H, and M. Terms with a software type belong to kind S, terms with a hardware value type belong to kind H, and terms with a hardware module type belong to kind M. More explicitly, all types of kind M are of type $T_H \to T_H$, where T_H is some type of kind H.