

Compilers Research

Aditya Srinivasan, Drew Hilton

Fall 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.

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, fn(i) => if(i % 3 = 0)
    then 'b:1
    else 'b:0]
```

```
#[32, fn(i) => case i of
    0 => 'b:0
  |: i => d[i - 1] & c[i]]
```

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 attaches certain properties to values.

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.
~1003.47e07
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 $\backslash\{a: \text{int}, b: \text{float}, c: (\text{int} * \text{int list})\}$.

The unit type is merely an alias for the empty tuple type. The syntax for 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 the functional type $(\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b list}$. Similarly, tuples can be passed as values, as demonstrated by the `#1` function, which has the functional type $(\text{'a} * \dots) \rightarrow \text{'a}$.

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. Bitwise operators belong in two categories: the first category corresponds to generic operators that can be applied to 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. The second category corresponds to specific operators that only operate on particular hardware value types. The first four operators below are generic, whereas the latter nine are specific. This is made clear in the paragraph below.

1. `&`: bitwise-and
2. `|`: bitwise-or
3. `^`: bitwise-xor
4. `!`: bitwise-complementation
5. `<<`: logical left shift
6. `>>`: logical right shift
7. `>>>`: arithmetic right shift
8. `&->`: and-reduction
9. `|->`: or-reduction
10. `^->`: xor-reduction
11. `&&`: and-collapse
12. `||`: or-collapse
13. `^^`: xor-collapse

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 next 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.

The next three operators are unary and have the functional type `bit[n] → bit`. Each of these applies the appropriate operator to each bit within the bit array in a fold-like manner, and return the final result.

Finally, the last three operators are binary and have the functional type `bit[n] → bit[n] → bit`. Each of these applies the corresponding reduction operation to each bit array, and then performs the corresponding bitwise operation to the resulting two bits.

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

The first operator has the type `int → int`. All of the remaining 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 an integer to a bit array of size `n` by interpreting the integer as unsigned. This function has type `int → 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 $(\text{'a} \rightarrow \text{'hd}) \rightarrow \text{'a list} \rightarrow \text{'hd}[n]$, 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 $\text{'a} \rightarrow \text{'a @ 1}$. 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 $(\text{'a} * \text{'b} * \text{'c}) \rightarrow (\text{'a @ 2} * \text{'b @ 2} * \text{'c @ 2})$. 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 $(\text{'a} * \text{'b} * \text{'c}) \rightarrow (\text{'a @ 1} * \text{'b @ 2} * \text{'c @ 3})$.

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 $(\text{'a} * \text{'b} * \text{'c}) \rightarrow (\text{'a @ ?} * \text{'b @ ?} * \text{'c @ 3})$.

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 $\text{'hv @ m} \rightarrow \text{'hv @ n}$. 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 be synchronized to have the same temporal type using the function `sync` which has type $(\text{'hd1 @ n} * \text{'hd2 @ m}) \rightarrow (\text{'hv1 @ t} * \text{'hv2 @ t})$ 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, opc) = alu(z, opc, 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 = F00 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 programming 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 Software 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.

4.1.1.1 Software Value Grammar

```

⟨swv⟩ ::= ⟨integer⟩
      | ⟨real⟩
      | ⟨string⟩
      | ⟨list⟩
      | ⟨sw-record⟩
      | ⟨sw⟩
      | ⟨ref⟩
      | ⟨sw-variant⟩
      | ⟨function⟩

⟨integer⟩ ::= i ∈ ℤ ∩ [-263, 263 - 1]

⟨real⟩ ::= r ∈ ℝ ∩ [2-1074, (2 - 2-52) × 21023] ∩ [-(2 - 2-52) × 21023, -2-1074] ∩ {numbers expressible as IEEE FP}

```

```

⟨string⟩ ::= s ∈ ⋃i=0263-1 Ai where A is the ASCII alphabet and Ai denotes a sequence of i characters from the alphabet A

⟨list⟩ ::= ⟨swv⟩ :: ⟨list⟩
      | nil

⟨sw-record⟩ ::= {li = ⟨swv⟩ii ∈ 1..n}

⟨sw⟩ ::= sw ⟨hwv⟩

⟨ref⟩ ::= ref ⟨swv⟩

⟨sw-variant⟩ ::= Ci ⟨swv⟩

⟨function⟩ ::= λx : TS.e

```

4.1.1.2 Software Term Grammar

```

⟨swt⟩ ::= ⟨sw-literal⟩
      | ⟨sw-access⟩
      | ⟨sw-let-binding⟩
      | ⟨conditional⟩
      | ⟨operations⟩
      | ⟨assign⟩
      | ⟨with⟩
      | ⟨pattern-match⟩
      | ⟨seq⟩
      | ⟨sw-apply⟩

⟨sw-literal⟩ ::= ⟨identifier⟩
      | ⟨integer-literal⟩
      | ⟨real-literal⟩
      | ⟨string-literal⟩
      | ⟨list-literal⟩
      | ⟨sw-record-literal⟩
      | ⟨ref-literal⟩
      | ⟨sw-literal⟩

⟨identifier⟩ ::= ⟨id-start⟩ ⟨id-tail⟩

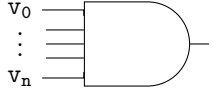
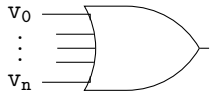
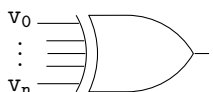
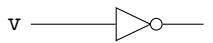
⟨id-start⟩ ::= {any alphabetic character or underscore}

⟨id-tail⟩ ::= ⟨id-tail⟩ {any alphanumeric character or underscore}
      | ε

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

```


$\langle \text{binary-integer} \rangle$	$::= \#'\mathbf{b}: \langle \text{binary-digits} \rangle$	$\langle \text{hex-digits} \rangle$	$::= \langle \text{hex-digits} \rangle \langle \text{hex-digit} \rangle$ $\langle \text{hex-digit} \rangle$
$\langle \text{octal-integer} \rangle$	$::= \#'\mathbf{o}: \langle \text{octal-digits} \rangle$	$\langle \text{sign} \rangle$	$::= \sim$
$\langle \text{decimal-integer} \rangle$	$::= \langle \text{decimal-digits} \rangle$ $\langle \text{sign} \rangle \langle \text{decimal-digits} \rangle$	$\langle \text{string-literal} \rangle$	$::= " \langle \text{string-tail} \rangle "$
$\langle \text{hex-integer} \rangle$	$::= \#'\mathbf{x}: \langle \text{hex-digits} \rangle$	$\langle \text{string-tail} \rangle$	$::= \langle \text{string-tail} \rangle \langle \text{string-character} \rangle$ ϵ
$\langle \text{integer-tail} \rangle$	$::= \langle \text{integer-tail} \rangle \langle \text{decimal-digit} \rangle$ $\langle \text{decimal-digit} \rangle$	$\langle \text{string-character} \rangle$	$::= \{\text{any printable character, including space, except for double-quotes (") and backslash (\)}\}$ $\backslash \langle \text{escape-character} \rangle$
$\langle \text{real-literal} \rangle$	$::= \langle \text{real-tail} \rangle$ $\langle \text{sign} \rangle \langle \text{real-tail} \rangle$	$\langle \text{escape-character} \rangle$	$::= \text{any of } \{\backslash, ', ", \text{a}, \text{b}, \text{e}, \text{f}, \text{n}, \text{r}, \text{t}, 0\}$ $\langle \text{hex-digits} \rangle$
$\langle \text{real-tail} \rangle$	$::= \langle \text{decimal-digits} \rangle .$ $\langle \text{decimal-digits-or-empty} \rangle$ $\langle \text{exponent-or-empty} \rangle$ $\langle \text{decimal-digits-or-empty} \rangle .$ $\langle \text{decimal-digits} \rangle$ $\langle \text{exponent-or-empty} \rangle$ $\langle \text{decimal-digits} \rangle \langle \text{exponent} \rangle$	$\langle \text{list-literal} \rangle$	$::= [\langle \text{list-body} \rangle]$ $\langle \text{swt} \rangle :: \langle \text{list-literal} \rangle$ nil
$\langle \text{decimal-digits-or-empty} \rangle$	$::= \langle \text{decimal-digits} \rangle$ ϵ	$\langle \text{list-body} \rangle$	$::= \langle \text{swt} \rangle \langle \text{list-body} \rangle$ $, \langle \text{list-body} \rangle$ ϵ
$\langle \text{exponent} \rangle$	$::= \mathbf{E} \langle \text{decimal-digits} \rangle$ $\mathbf{E} \langle \text{sign} \rangle \langle \text{decimal-digits} \rangle$ $\mathbf{e} \langle \text{decimal-digits} \rangle$ $\mathbf{e} \langle \text{sign} \rangle \langle \text{decimal-digits} \rangle$	$\langle \text{sw-record-literal} \rangle$	$::= \{ \langle \text{sw-record-body} \rangle \}$
$\langle \text{exponent-or-empty} \rangle$	$::= \langle \text{exponent} \rangle$ ϵ	$\langle \text{sw-record-body} \rangle$	$::= \langle \text{identifier} \rangle = \langle \text{swt} \rangle \langle \text{sw-record-body} \rangle$ $, \langle \text{sw-record-body} \rangle$ ϵ
$\langle \text{binary-digit} \rangle$	$::= \text{any of } \{0, 1\}$	$\langle \text{ref-literal} \rangle$	$::= \mathbf{ref} \langle \text{swt} \rangle$
$\langle \text{octal-digit} \rangle$	$::= \text{any of } \{0, 1, 2, 3, 4, 5, 6, 7\}$	$\langle \text{sw-literal} \rangle$	$::= \mathbf{sw} \langle \text{hwv} \rangle$
$\langle \text{decimal-digit} \rangle$	$::= \text{any of } \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$	$\langle \text{access} \rangle$	$::= \langle \text{struct-access} \rangle$ $\langle \text{sw-record-access} \rangle$ $\langle \text{ref-access} \rangle$
$\langle \text{hex-digit} \rangle$	$::= \text{any of } \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{a}, \text{b}, \text{c}, \text{d}, \text{e}, \text{f}, \text{A}, \text{B}, \text{C}, \text{D}, \text{E}, \text{F}\}$	$\langle \text{struct-access} \rangle$	$::= \langle \text{identifier} \rangle . \langle \text{identifier} \rangle$
$\langle \text{binary-digits} \rangle$	$::= \langle \text{binary-digits} \rangle \langle \text{binary-digit} \rangle$ $\langle \text{binary-digit} \rangle$	$\langle \text{sw-record-access} \rangle$	$::= \# \langle \text{identifier} \rangle \langle \text{swt} \rangle$
$\langle \text{octal-digits} \rangle$	$::= \langle \text{octal-digits} \rangle \langle \text{octal-digit} \rangle$ $\langle \text{octal-digit} \rangle$	$\langle \text{ref-access} \rangle$	$::= \$ \langle \text{swt} \rangle$
$\langle \text{decimal-digits} \rangle$	$::= \langle \text{decimal-digits} \rangle \langle \text{decimal-digit} \rangle$ $\langle \text{decimal-digit} \rangle$	$\langle \text{sw-let-binding} \rangle$	$::= \mathbf{let} \langle \text{sw-decs} \rangle \mathbf{in} \langle \text{swt} \rangle \mathbf{end}$
		$\langle \text{sw-decs} \rangle$	$::= \langle \text{sw-val-dec} \rangle \langle \text{sw-decs} \rangle$ $\langle \text{ty-dec} \rangle \langle \text{sw-decs} \rangle$ $\langle \text{datatype-dec} \rangle \langle \text{sw-decs} \rangle$ $\langle \text{fun-dec} \rangle \langle \text{sw-decs} \rangle$ ϵ

$\langle sw\text{-}val\text{-}dec \rangle$	$::= \text{val } \langle identifier \rangle = \langle swt \rangle$ $\text{val } \langle identifier \rangle : \langle ty \rangle = \langle swt \rangle$	$\langle int\text{-}op \rangle$	$::= \sim \langle swt \rangle$ $\langle swt \rangle + \langle swt \rangle$ $\langle swt \rangle - \langle swt \rangle$ $\langle swt \rangle * \langle swt \rangle$ $\langle swt \rangle / \langle swt \rangle$ $\langle swt \rangle \% \langle swt \rangle$
$\langle ty\text{-}dec \rangle$	$::= \text{type } \langle identifier \rangle = \langle ty \rangle$	$\langle real\text{-}op \rangle$	$::= \langle swt \rangle +. \langle swt \rangle$ $\langle swt \rangle -. \langle swt \rangle$ $\langle swt \rangle *. \langle swt \rangle$ $\langle swt \rangle /. \langle swt \rangle$
$\langle ty \rangle$	$::= \langle tid \rangle$ $\langle identifier \rangle$ $\{ \langle ty\text{-}fields \rangle \}$ $\# \{ \langle ty\text{-}fields \rangle \}$ $\langle ty \rangle @ \langle swt \rangle$ $\langle ty \rangle [\langle swt \rangle]$ $\langle ty \rangle \text{ list}$ $\langle ty \rangle \text{ ref}$ $\langle ty \rangle \text{ sw}$ $\langle ty \rangle \rightarrow \langle ty \rangle$ $(\langle ty \rangle)$	$\langle compare\text{-}op \rangle$	$::= \langle swt \rangle = \langle sw\text{-}t \rangle$ $\langle swt \rangle <> \langle swt \rangle$ $\langle swt \rangle > \langle swt \rangle$ $\langle swt \rangle < \langle swt \rangle$ $\langle swt \rangle >= \langle swt \rangle$ $\langle swt \rangle <= \langle swt \rangle$
$\langle tid \rangle$	$::= ' \langle identifier \rangle$	$\langle assign \rangle$	$::= \langle swt \rangle := \langle swt \rangle$
$\langle ty\text{-}fields \rangle$	$::= \langle identifier \rangle : \langle ty \rangle \langle ty\text{-}fields\text{-}tail \rangle$	$\langle with \rangle$	$::= \langle swt \rangle \text{ with } \langle swt \rangle$
$\langle ty\text{-}fields\text{-}tail \rangle$	$::= , \langle ty\text{-}fields \rangle$ ϵ	$\langle pattern\text{-}match \rangle$	$::= \text{case } \langle swt \rangle \text{ of } \langle cases \rangle$
$\langle dataty\text{-}dec \rangle$	$::= \text{datatype } \langle identifier \rangle =$ $\langle identifier \rangle \text{ of } \langle ty \rangle \langle dataty\text{-}tail \rangle$	$\langle cases \rangle$	$::= \langle swt \rangle \Rightarrow \langle swt \rangle \langle case\text{-}tail \rangle$
$\langle dataty\text{-}tail \rangle$	$::= : \langle identifier \rangle \text{ of } \langle ty \rangle$ $\langle dataty\text{-}tail \rangle$ ϵ	$\langle case\text{-}tail \rangle$	$::= \langle case\text{-}tail \rangle : \langle swt \rangle \Rightarrow \langle swt \rangle$ ϵ
$\langle fun\text{-}dec \rangle$	$::= \text{fun } \langle identifier \rangle \langle fun\text{-}params \rangle =$ $\langle swt \rangle$ $\text{fun } \langle identifier \rangle \langle fun\text{-}params \rangle :$ $\langle ty \rangle = \langle swt \rangle$	$\langle seq \rangle$	$::= (\langle swt \rangle)$ $(\langle swt \rangle ; \langle swt \rangle \langle seq\text{-}tail \rangle)$
$\langle fun\text{-}params \rangle$	$::= \langle fun\text{-}params \rangle \langle fun\text{-}param \rangle$ $\langle fun\text{-}param \rangle$	$\langle seq\text{-}tail \rangle$	$::= ; \langle swt \rangle \langle seq\text{-}tail \rangle$ ϵ
$\langle fun\text{-}param \rangle$	$::= \langle identifier \rangle$ $(\langle no\text{-}ty\text{-}fields \rangle)$ $(\langle ty\text{-}fields \rangle)$	$\langle sw\text{-}apply \rangle$	$::= \langle swt \rangle \langle swt \rangle$
$\langle no\text{-}ty\text{-}fields \rangle$	$::= \langle identifier \rangle$ $\langle no\text{-}ty\text{-}fields \rangle , \langle identifier \rangle$	4.1.2 Hardware Grammar	
$\langle conditional \rangle$	$::= \text{if } \langle swt \rangle \text{ then } \langle swt \rangle \text{ else } \langle swt \rangle$ $\text{if } \langle swt \rangle \text{ then } \langle swt \rangle$	4.1.2.1 Hardware Value Grammar	
$\langle operations \rangle$	$::= \langle arith\text{-}op \rangle$ $\langle compare\text{-}op \rangle$ $\langle list\text{-}op \rangle$	$\langle hwv \rangle$	$::= 0$ 1     $\# [v_i^{i \in 0..n-1}]$
$\langle arith\text{-}op \rangle$	$::= \langle int\text{-}op \rangle$ $\langle real\text{-}op \rangle$		

$$\begin{array}{l}
| \{ l_i = v_i^{i \in 1..n} \} \\
| v_1 << v_2 \\
| v_1 >> v_2 \\
| v_1 >>> v_2 \\
| \text{dff}(v)
\end{array}$$

4.1.2.2 Hardware Value Syntax

$$\begin{array}{ll}
\langle hwv \rangle & ::= \langle hw\text{-literal} \rangle \\
& | \langle hw\text{-access} \rangle \\
& | \langle hw\text{-conditional} \rangle \\
& | \langle hw\text{-pattern-match} \rangle \\
& | \langle hw\text{-let-binding} \rangle \\
& | \langle hw\text{-variant} \rangle \\
& | \langle bit\text{-op} \rangle \\
& | \langle hw\text{-seq} \rangle \\
& | \langle hw\text{-apply} \rangle \\
\\
\langle hw\text{-literal} \rangle & ::= \langle identifier \rangle \\
& | \langle bit \rangle \\
& | \langle array\text{-literal} \rangle \\
& | \langle hw\text{-record-literal} \rangle \\
& | \langle hw\text{-module} \rangle \\
\\
\langle bit \rangle & ::= 'b: \langle binary\text{-digit} \rangle \\
\\
\langle array\text{-literal} \rangle & ::= \# [\langle array\text{-body} \rangle] \\
& | \langle inline\text{-array} \rangle \\
& | \langle bit\text{-array} \rangle \\
\\
\langle array\text{-body} \rangle & ::= \langle hwv \rangle \langle array\text{-body} \rangle \\
& | , \langle array\text{-body} \rangle \\
& | \epsilon \\
\\
\langle inline\text{-array} \rangle & ::= \# [\langle swt \rangle ; \langle swt \rangle] \\
\\
\langle bit\text{-array-inst} \rangle & ::= \langle swt \rangle \langle tid \rangle : \langle swt \rangle \\
\\
\langle hw\text{-record-literal} \rangle & ::= \# \{ \langle hw\text{-record-body} \rangle \} \\
\\
\langle hw\text{-record-body} \rangle & ::= \langle identifier \rangle = \langle hwv \rangle \\
& | \langle hw\text{-record-body} \rangle \\
& | , \langle hw\text{-record-body} \rangle \\
& | \epsilon \\
\\
\langle hw\text{-module} \rangle & ::= \lambda x : T_H.e \\
\\
\langle hw\text{-access} \rangle & ::= \langle array\text{-access} \rangle \\
& | \langle hw\text{-record-access} \rangle \\
\\
\langle array\text{-access} \rangle & ::= \langle hwv \rangle [\langle swt \rangle] \\
\\
\langle hw\text{-record-access} \rangle & ::= \# \langle identifier \rangle \langle hwv \rangle
\end{array}$$

$$\begin{array}{ll}
\langle hw\text{-conditional} \rangle & ::= \text{if } \langle swt \rangle \text{ then } \langle hwv \rangle \text{ else } \langle hwv \rangle \\
\\
\langle hw\text{-pattern-match} \rangle & ::= \text{case } \langle hwv \rangle \text{ of } \langle hw\text{-cases} \rangle \\
\\
\langle hw\text{-cases} \rangle & ::= \langle hwv \rangle \Rightarrow \langle hwv \rangle \langle hw\text{-case-tail} \rangle \\
\\
\langle hw\text{-case-tail} \rangle & ::= \langle hw\text{-case-tail} \rangle \quad | : \langle hwv \rangle \Rightarrow \langle hwv \rangle \\
& | \epsilon \\
\\
\langle hw\text{-let-binding} \rangle & ::= \text{let } \langle hw\text{-decs} \rangle \text{ in } \langle hwv \rangle \text{ end} \\
\\
\langle hw\text{-decs} \rangle & ::= \langle hw\text{-val-dec} \rangle \langle hw\text{-decs} \rangle \\
& | \langle module\text{-dec} \rangle \langle hw\text{-decs} \rangle \\
& | \epsilon \\
\\
\langle hw\text{-val-dec} \rangle & ::= \text{val } \langle identifier \rangle = \langle hwv \rangle \\
& | \text{val } \langle identifier \rangle : \langle ty \rangle = \langle hwv \rangle \\
\\
\langle module\text{-dec} \rangle & ::= \text{module } \langle identifier \rangle \langle fun\text{-param} \rangle \\
& = \langle hwv \rangle \\
& | \text{module } \langle identifier \rangle \langle fun\text{-param} \rangle \\
& : \langle ty \rangle = \langle hwv \rangle \\
\\
\langle hw\text{-variant} \rangle & ::= \langle C_i = \langle hwv \rangle \rangle \\
\\
\langle bit\text{-op} \rangle & ::= ! \langle hwv \rangle \\
& | \langle hwv \rangle \mapsto \langle hwv \rangle \\
& | \langle hwv \rangle \&\rightarrow \langle hwv \rangle \\
& | \langle hwv \rangle \wedge\rightarrow \langle hwv \rangle \\
& | \langle hwv \rangle \&\& \langle hwv \rangle \\
& | \langle hwv \rangle || \langle hwv \rangle \\
& | \langle hwv \rangle \wedge\wedge \langle hwv \rangle \\
& | \langle hwv \rangle \& \langle hwv \rangle \\
& | \langle hwv \rangle | \langle hwv \rangle \\
& | \langle hwv \rangle \wedge \langle hwv \rangle \\
& | \langle hwv \rangle << \langle hwv \rangle \\
& | \langle hwv \rangle >> \langle hwv \rangle \\
& | \langle hwv \rangle >>> \langle hwv \rangle \\
\\
\langle hw\text{-seq} \rangle & ::= (\langle hwv \rangle) \\
& | (\langle hwv \rangle ; \langle hwv \rangle \langle hw\text{-seq-tail} \rangle) \\
\\
\langle hw\text{-seq-tail} \rangle & ::= ; \langle hwv \rangle \langle hw\text{-seq-tail} \rangle \\
& | \epsilon \\
\\
\langle hw\text{-apply} \rangle & ::= \langle hwv \rangle \langle hwv \rangle
\end{array}$$

4.1.2.3 Hardware Term Grammar

Hardware data is always manifest as a terminal value, and there are no steps that can be taken in order to evaluate hardware data any further. As such, the concept of a "term" does not exist in hardware and our hardware term grammar is empty.

4.1.3 Derived Term Grammar

We define additional terms as formulations of existing terms already defined. These are, in a sense, derived from existing terms and as such as omitted from proofs since proving correctness in the case of the original term guarantees correctness in the case of the derived term.

$$(t_i)^{i \in 1..n} \stackrel{\text{def}}{=} \{i = t_i\}^{i \in 1..n} \quad (\text{tuple})$$

$$() \stackrel{\text{def}}{=} \{\} \quad (\text{unit})$$

$$t_1 \text{ andalso } t_2 \stackrel{\text{def}}{=} \text{if } t_1 \text{ then } t_2 \text{ else } 0 \quad (\text{andalso})$$

$$t_1 \text{ orelse } t_2 \stackrel{\text{def}}{=} \text{if } t_1 \text{ then } 1 \text{ else } t_2 \quad (\text{orelse})$$

$$\text{not } t_1 \stackrel{\text{def}}{=} \text{if } t_1 \text{ then } 0 \text{ else } 1 \quad (\text{not})$$

$$\frac{t_1 : \text{bit} \quad t_2 : \text{bit}}{t_1 \ \& \ t_2 \stackrel{\text{def}}{=} \&->\#[t_1, t_2]} \quad (\text{bitwise-and})$$

$$\frac{t_1 : \text{bit} \quad t_2 : \text{bit}}{t_1 \ | \ t_2 \stackrel{\text{def}}{=} |->\#[t_1, t_2]} \quad (\text{bitwise-or})$$

$$\frac{t_1 : \text{bit} \quad t_2 : \text{bit}}{t_1 \ \wedge \ t_2 \stackrel{\text{def}}{=} \wedge->\#[t_1, t_2]} \quad (\text{bitwise-xor})$$

$$t_1 \ \&\& \ t_2 \stackrel{\text{def}}{=} |->t_1 \ \& \ |->t_2 \quad (\text{logical-and})$$

$$t_1 \ || \ t_2 \stackrel{\text{def}}{=} |->t_1 \ | \ |->t_2 \quad (\text{logical-or})$$

$$t_1 \ \wedge\wedge \ t_2 \stackrel{\text{def}}{=} |->t_1 \ \wedge \ |->t_2 \quad (\text{logical-xor})$$

$$\text{if } t_1 \text{ then } t_2 \stackrel{\text{def}}{=} \text{if } t_1 \text{ then } t_2 \text{ else } \{\} \quad (\text{if-then})$$

$$t_1; t_2 \stackrel{\text{def}}{=} (\lambda x : T. t_2) \ t_1 \quad (\text{sequence})$$

where $x \notin FV(t_2)$

4.1.4 Type Grammar

$$\begin{aligned} \langle S \rangle &::= \text{int} \\ &| \text{real} \\ &| \text{string} \\ &| T_H \text{ sw} \\ &| T_M \text{ sw} \\ &| T_S \text{ list} \\ &| \{l_i : T_{Si}\}^{i \in 1..n} \\ &| T_S \text{ ref} \\ &| \langle C_i : T_i \rangle^{i \in 1..n} \\ &| T_S \rightarrow T_S \end{aligned}$$

$$\begin{aligned} \langle H \rangle &::= \text{bit} \\ &| T_H [\text{int}] \\ &| \#\{l_i : T_{Hi}\}^{i \in 1..n} \\ &| T_H @ \text{int} \end{aligned}$$

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

4.1.5 Typing Rules

Next, we introduce several typing rules. These include axioms indicating the types of certain values, and inferences for the typing restrictions and results of terms.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR})$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2} \quad (\text{T-APP})$$

$$< \text{int} > : \text{int} \quad (\text{T-INT})$$

$$< \text{real} > : \text{real} \quad (\text{T-REAL})$$

$\langle \text{string} \rangle : \text{string}$	(T-STRING)	$\frac{t_1 : \text{real} \quad t_2 : \text{real}}{t_1 /. t_2 : \text{real}}$	(T-REAL-DIV)
$\langle \text{bit} \rangle : \text{bit}$	(T-BIT)	$\frac{t_1 : T_H}{!t_1 : T_H}$	(T-BIT-NEG)
$\text{nil} : T_S \text{ list}$	(T-NIL)	$\frac{t_1 : T_H \quad t_2 : T_H}{t_1 \& t_2 : T_H}$	(T-AND)
$\frac{t_1 : \text{int}}{\sim t_1 : \text{int}}$	(T-INT-NEG)	$\frac{t_1 : T_H \quad t_2 : T_H}{t_1 t_2 : T_H}$	(T-OR)
$\frac{t_1 : \text{int} \quad t_2 : \text{int}}{t_1 + t_2 : \text{int}}$	(T-INT-ADD)	$\frac{t_1 : T_H \quad t_2 : T_H}{t_1 \wedge t_2 : T_H}$	(T-XOR)
$\frac{t_1 : \text{int} \quad t_2 : \text{int}}{t_1 - t_2 : \text{int}}$	(T-INT-SUB)	$\frac{t_1 : \text{bit}[n]}{\&-> t_1 : \text{bit}}$	(T-AND-RED)
$\frac{t_1 : \text{int} \quad t_2 : \text{int}}{t_1 * t_2 : \text{int}}$	(T-INT-MUL)	$\frac{t_1 : \text{bit}[n]}{ -> t_1 : \text{bit}}$	(T-OR-RED)
$\frac{t_1 : \text{int} \quad t_2 : \text{int}}{t_1 / t_2 : \text{int}}$	(T-INT-DIV)	$\frac{t_1 : \text{bit}[n]}{\wedge-> t_1 : \text{bit}}$	(T-XOR-RED)
$\frac{t_1 : \text{int} \quad t_2 : \text{int}}{t_1 \% t_2 : \text{int}}$	(T-INT-MOD)	$\frac{t_1 : \text{bit}[n] \quad t_2 : \text{bit}[n]}{t_1 \&\& t_2 : \text{bit}}$	(T-LOG-AND)
$\frac{t_1 : \text{real}}{\sim t_1 : \text{real}}$	(T-REAL-NEG)	$\frac{t_1 : \text{bit}[n] \quad t_2 : \text{bit}[n]}{t_1 t_2 : \text{bit}}$	(T-LOG-OR)
$\frac{t_1 : \text{real} \quad t_2 : \text{real}}{t_1 +. t_2 : \text{real}}$	(T-REAL-ADD)	$\frac{t_1 : \text{bit}[n] \quad t_2 : \text{bit}[n]}{t_1 \wedge \wedge t_2 : \text{bit}}$	(T-LOG-XOR)
$\frac{t_1 : \text{real} \quad t_2 : \text{real}}{t_1 -. t_2 : \text{real}}$	(T-REAL-SUB)	$\frac{t_1 : \text{bit}[n] \quad t_2 : \text{bit}[m]}{t_1 << t_2 : \text{bit}[n]}$	(T-SLL)
$\frac{t_1 : \text{real} \quad t_2 : \text{real}}{t_1 *. t_2 : \text{real}}$	(T-REAL-MUL)	$\frac{t_1 : \text{bit}[n] \quad t_2 : \text{bit}[m]}{t_1 >> t_2 : \text{bit}[n]}$	(T-SRL)
		$\frac{t_1 : \text{bit}[n] \quad t_2 : \text{bit}[m]}{t_1 >>> t_2 : \text{bit}[n]}$	(T-SRA)

$\frac{t_1 : T_S \quad t_2 : T_S}{t_1 = t_2 : \text{int}}$	(T-EQ)	$\frac{t_1 : T_S \text{ ref} \quad t_2 : T_S}{t_1 := t_2 : \text{unit}}$	(T-ASSIGN)
$\frac{t_1 : T_S \quad t_2 : T_S}{t_1 <> t_2 : \text{int}}$	(T-NEQ)		
$\frac{t_1 : T_S \quad t_2 : T_S}{t_1 >= t_2 : \text{int}}$	(T-GEQ)	$\frac{t_1 : T_H[n] \quad t_2 : \text{int}}{t_1[t_2] : T_H}$	(T-ARR-ACC)
$\frac{t_1 : T_S \quad t_2 : T_S}{t_1 > t_2 : \text{int}}$	(T-GT)	$\frac{t_1 : T_S \text{ ref}}{\$t_1 : T_S}$	(T-DEREF)
$\frac{t_1 : T_S \quad t_2 : T_S}{t_1 <= t_2 : \text{int}}$	(T-LEQ)		
$\frac{t_1 : T_S \quad t_2 : T_S}{t_1 < t_2 : \text{int}}$	(T-LT)	$\frac{\text{for each } i \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{i \in 1..n}\} : \{l_i : T_i^{i \in 1..n}\}}$	(T-RCD)
$\frac{t_1 : T_S \quad t_2 : T_S \text{ list}}{t_1 :: t_2 : T_S \text{ list}}$	(T-CONS)	$\frac{\Gamma \vdash t_1 : \{l_i : T_i^{i \in 1..n}\}}{\Gamma \vdash \#l_j t_1 : T_j}$	(T-PROJ)
$\frac{t_1 : \text{int} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$	(T-IFELSE)		
$\frac{t_1 : T_S}{\text{ref } t_1 : T_S \text{ ref}}$	(T-REF)	$\frac{\Gamma, \tau \vdash t_1 : T_1 \quad \Gamma, \tau, x_1 : T_1 \vdash \text{let } (x_1 = t_1)^{i \in 2..n} \text{ in } t_0 \text{ end} : T_0^{(T-LET)}}{\text{let } (x_1 = t_1)^{i \in 1..n} \text{ in } t_0 \text{ end} : T_0}$	(T-LET)
$\frac{t_1 : T_H}{\text{sw } t_1 : T_H \text{ sw}}$	(T-SW-H)		
$\frac{t_1 : T_M}{\text{sw } t_1 : T_M \text{ sw}}$	(T-SW-M)	$\frac{D : \langle C_i : T_i \rangle^{i \in 1..n} \in \tau \quad \Gamma \vdash t : T_j}{\Gamma, \tau \vdash C_j t : D}$	(T-DATATY)
$\frac{t_1 : T_H \text{ sw}}{\text{hw } t_1 : T_H}$	(T-HW-H)		
$\frac{t_1 : T_M \text{ sw}}{\text{hw } t_1 : T_M}$	(T-HW-M)	$\frac{\Gamma \vdash t_0 : \langle C_i : T_i \rangle^{i \in 1..n} \quad \text{for each } i \Gamma, x_i : T_i \vdash t_i : T}{\Gamma \vdash \text{case } t_0 \text{ of } C_i x_i \Rightarrow t_i^{i \in 1..n} : T}$	(T-CASE)

4.1.6 Evaluation Rules

$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \quad (\text{E-APP1})$	$\frac{t_2 \longrightarrow t'_2}{v_1 * t_2 \longrightarrow v_1 * t'_2} \quad (\text{E-INT-MUL2})$
$\frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2} \quad (\text{E-APP2})$	$\frac{t_1 \longrightarrow t'_1}{t_1 / t_2 \longrightarrow t'_1 / t_2} \quad (\text{E-INT-DIV1})$
$(\lambda x. t_1) v_1 \longrightarrow [x \mapsto v_1] t_1 \quad (\text{E-APPABS})$	$\frac{t_2 \longrightarrow t'_2}{v_1 / t_2 \longrightarrow v_1 / t'_2} \quad (\text{E-INT-DIV2})$
$\frac{t_1 \longrightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3} \quad (\text{E-IFELSE})$	$\frac{t_1 \longrightarrow t'_1}{t_1 \% t_2 \longrightarrow t'_1 \% t_2} \quad (\text{E-INT-MOD1})$
$\frac{v_1 : \text{int} \quad v_1 \neq 0}{\text{if } v_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow t_2} \quad (\text{E-IFELSE-T})$	$\frac{t_2 \longrightarrow t'_2}{v_1 \% t_2 \longrightarrow v_1 \% t'_2} \quad (\text{E-INT-MOD2})$
$\text{if } 0 \text{ then } t_2 \text{ else } t_3 \longrightarrow t_3 \quad (\text{E-IFELSE-F})$	$\frac{t_1 \longrightarrow t'_1}{t_1 +. t_2 \longrightarrow t'_1 +. t_2} \quad (\text{E-REAL-ADD1})$
$\frac{t_1 \longrightarrow t'_1}{\sim t_1 \longrightarrow \sim t'_1} \quad (\text{E-NEG})$	$\frac{t_2 \longrightarrow t'_2}{v_1 +. t_2 \longrightarrow v_1 +. t'_2} \quad (\text{E-REAL-ADD2})$
$\frac{t_1 \longrightarrow t'_1}{t_1 + t_2 \longrightarrow t'_1 + t_2} \quad (\text{E-INT-ADD1})$	$\frac{t_1 \longrightarrow t'_1}{t_1 -. t_2 \longrightarrow t'_1 -. t_2} \quad (\text{E-REAL-SUB1})$
$\frac{t_2 \longrightarrow t'_2}{v_1 + t_2 \longrightarrow v_1 + t'_2} \quad (\text{E-INT-ADD2})$	$\frac{t_2 \longrightarrow t'_2}{v_1 -. t_2 \longrightarrow v_1 -. t'_2} \quad (\text{E-REAL-SUB2})$
$\frac{t_1 \longrightarrow t'_1}{t_1 - t_2 \longrightarrow t'_1 - t_2} \quad (\text{E-INT-SUB1})$	$\frac{t_1 \longrightarrow t'_1}{t_1 *. t_2 \longrightarrow t'_1 *. t_2} \quad (\text{E-REAL-MUL1})$
$\frac{t_2 \longrightarrow t'_2}{v_1 - t_2 \longrightarrow v_1 - t'_2} \quad (\text{E-INT-SUB2})$	$\frac{t_2 \longrightarrow t'_2}{v_1 *. t_2 \longrightarrow v_1 *. t'_2} \quad (\text{E-REAL-MUL2})$
$\frac{t_1 \longrightarrow t'_1}{t_1 * t_2 \longrightarrow t'_1 * t_2} \quad (\text{E-INT-MUL1})$	$\frac{t_1 \longrightarrow t'_1}{t_1 /. t_2 \longrightarrow t'_1 /. t_2} \quad (\text{E-REAL-DIV1})$
	$\frac{t_2 \longrightarrow t'_2}{v_1 /. t_2 \longrightarrow v_1 /. t'_2} \quad (\text{E-REAL-DIV2})$

$$\frac{t_j \longrightarrow t'_j}{\# [v_i^{i \in 1..j-1}, t_j, t_k^{k \in j+1..n}] \longrightarrow \# [v_i^{i \in 1..j-1}, t'_j, t_k^{k \in j+1..n}]} \quad (\text{E-ARR})$$

$$\frac{t_1 \longrightarrow t'_1}{!t_1 \longrightarrow !t'_1} \quad (\text{E-BIT-NEG1})$$

$$\frac{v : T_H[n]}{!v \longrightarrow \# [!v[i]]^{i \in 0..n-1}} \quad (\text{E-BIT-NEG2})$$

$$\frac{v : \# \{l_i : T_{H_i}^{i \in 1..n}\}}{!v \longrightarrow \# \{l_i = !\#l_i \ v\}^{i \in 1..n}} \quad (\text{E-BIT-NEG3})$$

$$\frac{v : \text{bit}}{!v \longrightarrow v \text{ --- } \triangle \text{ --- } \bar{v}_1} \quad (\text{E-BIT-NEG})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ \& \ t_2 \longrightarrow t'_1 \ \& \ t_2} \quad (\text{E-AND1})$$

$$\frac{t_2 \longrightarrow t'_2}{v_1 \ \& \ t_2 \longrightarrow v_1 \ \& \ t'_2} \quad (\text{E-AND2})$$

$$\frac{v_1 : \text{bit}[n] \quad v_2 : \text{bit}[n]}{v_1 \ \& \ v_2 \longrightarrow \# [v_{1,i} \ \& \ v_{2,i}]^{i \in 0..n-1}} \quad (\text{E-AND3})$$

$$\frac{v_1 : \# \{l_i : T_{H_i}^{i \in 1..n}\} \quad v_2 : \# \{l_i : T_{H_i}^{i \in 1..n}\}}{v_1 \ \& \ v_2 \longrightarrow \# \{l_i = \#l_i \ v_1 \ \& \ \#l_i \ v_2\}^{i \in 1..n}} \quad (\text{E-AND4})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ | \ t_2 \longrightarrow t'_1 \ | \ t_2} \quad (\text{E-OR1})$$

$$\frac{t_2 \longrightarrow t'_2}{v_1 \ | \ t_2 \longrightarrow v_1 \ | \ t'_2} \quad (\text{E-OR2})$$

$$\frac{v_1 : \text{bit}[n] \quad v_2 : \text{bit}[n]}{v_1 \ | \ v_2 \longrightarrow \# [v_{1,i} \ | \ v_{2,i}]^{i \in 0..n-1}} \quad (\text{E-OR3})$$

$$\frac{v_1 : \# \{l_i : T_{H_i}^{i \in 1..n}\} \quad v_2 : \# \{l_i : T_{H_i}^{i \in 1..n}\}}{v_1 \ | \ v_2 \longrightarrow \# \{l_i = \#l_i \ v_1 \ | \ \#l_i \ v_2\}^{i \in 1..n}} \quad (\text{E-OR4})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ \wedge \ t_2 \longrightarrow t'_1 \ \wedge \ t_2} \quad (\text{E-XOR1})$$

$$\frac{t_2 \longrightarrow t'_2}{v_1 \ \wedge \ t_2 \longrightarrow v_1 \ \wedge \ t'_2} \quad (\text{E-XOR2})$$

$$\frac{v_1 : \text{bit}[n] \quad v_2 : \text{bit}[n]}{v_1 \ \wedge \ v_2 \longrightarrow \# [v_{1,i} \ \wedge \ v_{2,i}]^{i \in 0..n-1}} \quad (\text{E-XOR3})$$

$$\frac{v_1 : \# \{l_i : T_{H_i}^{i \in 1..n}\} \quad v_2 : \# \{l_i : T_{H_i}^{i \in 1..n}\}}{v_1 \ \wedge \ v_2 \longrightarrow \# \{l_i = \#l_i \ v_1 \ \wedge \ \#l_i \ v_2\}^{i \in 1..n}} \quad (\text{E-XOR4})$$

$$\frac{t_1 \longrightarrow t'_1}{\&->t_1 \longrightarrow \&->t'_1} \quad (\text{E-AND-RED1})$$

$$\frac{v : \text{bit}[n]}{\&->v \longrightarrow \begin{array}{c} v[0] \\ \vdots \\ v[n-1] \end{array} \text{ --- } \text{AND}} \quad (\text{E-AND-RED})$$

$$\frac{t_1 \longrightarrow t'_1}{|->t_1 \longrightarrow |->t'_1} \quad (\text{E-OR-RED1})$$

$$\frac{v : \text{bit}[n]}{|->v \longrightarrow \begin{array}{c} v[0] \\ \vdots \\ v[n-1] \end{array} \text{ --- } \text{OR}} \quad (\text{E-OR-RED})$$

$$\frac{t_1 \longrightarrow t'_1}{\wedge->t_1 \longrightarrow \wedge->t'_1} \quad (\text{E-XOR-RED1})$$

$$\begin{array}{c}
\hline v : \text{bit}[n] \\
\hline
\begin{array}{c}
v[0] \\
\vdots \\
v[n-1]
\end{array}
\begin{array}{c}
\text{---} \\
\text{---} \\
\text{---} \\
\text{---}
\end{array}
\begin{array}{c}
\text{---} \\
\text{---} \\
\text{---} \\
\text{---}
\end{array}
\end{array}
\quad (\text{E-XOR-RED})$$

$\wedge \rightarrow v \rightarrow$

$$\frac{t_1 \rightarrow t'_1}{t_1 << t_2 \rightarrow t'_1 << t_2} \quad (\text{E-SLL1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 << t_2 \rightarrow v_1 << t'_2} \quad (\text{E-SLL2})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 >> t_2 \rightarrow t'_1 >> t_2} \quad (\text{E-SRL1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 >> t_2 \rightarrow v_1 >> t'_2} \quad (\text{E-SRL2})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 >>> t_2 \rightarrow t'_1 >>> t_2} \quad (\text{E-SRA1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 >>> t_2 \rightarrow v_1 >>> t'_2} \quad (\text{E-SRA2})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 = t_2 \rightarrow t'_1 = t_2} \quad (\text{E-EQ1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 = t_2 \rightarrow v_1 = t'_2} \quad (\text{E-EQ2})$$

$$v_1 = v_2 \rightarrow \text{subject to semantics of type} \quad (\text{E-EQ})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 <> t_2 \rightarrow t'_1 <> t_2} \quad (\text{E-NEQ1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 <> t_2 \rightarrow v_1 <> t'_2} \quad (\text{E-NEQ2})$$

$$v_1 <> v_2 \rightarrow \text{subject to semantics of type} \quad (\text{E-NEQ})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 < t_2 \rightarrow t'_1 < t_2} \quad (\text{E-LT1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 < t_2 \rightarrow v_1 < t'_2} \quad (\text{E-LT2})$$

$$v_1 < v_2 \rightarrow \text{subject to semantics of type} \quad (\text{E-LT})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 \leq t_2 \rightarrow t'_1 \leq t_2} \quad (\text{E-LEQ1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 \leq t_2 \rightarrow v_1 \leq t'_2} \quad (\text{E-LEQ2})$$

$$v_1 \leq v_2 \rightarrow \text{subject to semantics of type} \quad (\text{E-LEQ})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 > t_2 \rightarrow t'_1 > t_2} \quad (\text{E-GT1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 > t_2 \rightarrow v_1 > t'_2} \quad (\text{E-GT2})$$

$$v_1 > v_2 \rightarrow \text{subject to semantics of type} \quad (\text{E-GT})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 \geq t_2 \rightarrow t'_1 \geq t_2} \quad (\text{E-GEQ1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 \geq t_2 \rightarrow v_1 \geq t'_2} \quad (\text{E-GEQ2})$$

$$v_1 \geq v_2 \rightarrow \text{subject to semantics of type} \quad (\text{E-GEQ})$$

$\frac{t_1 \longrightarrow t'_1}{t_1 :: t_2 \longrightarrow t'_1 :: t_2} \quad (\text{E-CONS1})$	$\frac{l \notin \text{dom}(\mu)}{\text{ref } v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \quad (\text{E-REFV})$
$\frac{t_2 \longrightarrow t'_2}{v_1 :: t_2 \longrightarrow v_1 :: t'_2} \quad (\text{E-CONS2})$	$\begin{array}{l} \text{let } x_1 = v_1 \ (x_i = t_i)^{i \in 2..n} \\ \text{in } t_0 \text{ end} \mid (\Gamma, \tau) \\ \longrightarrow \text{let } x_2 = t_2 \ (x_i = t_i)^{i \in 3..n} \\ \text{in } t_0 \text{ end} \mid (\Gamma, \tau, x_1 \mapsto v_1) \end{array} \quad (\text{E-LETV1})$
$\#l_j \ \{l_i = v_i^{i \in 1..n}\} \longrightarrow v_j \quad (\text{E-PROJ-RCD})$	$\text{let } x = v \text{ in } t \longrightarrow [x \mapsto v] t \quad (\text{E-LETV2})$
$\frac{t_1 \longrightarrow t'_1}{\#l \ t_1 \longrightarrow \#l \ t'_1} \quad (\text{E-PROJ})$	$\frac{t_1 \longrightarrow t'_1}{\text{let } x_1 = t_1 \ (x_i = t_i)^{i \in 2..n} \text{ in } t_0 \text{ end}} \quad (\text{E-LET})$
$\frac{t_j \longrightarrow t'_j}{\{l_i = v_i^{i \in 1..j-1}, l_j = t_j, l_k = t_k^{k \in j+1..n}\} \longrightarrow \{l_i = v_i^{i \in 1..j-1}, l_j = t'_j, l_k = t_k^{k \in j+1..n}\}} \quad (\text{E-RCD})$	$\frac{t_i \longrightarrow t'_i}{C_i \ t_i \longrightarrow C_i \ t'_i} \quad (\text{E-DATATY})$
$\{l_i = v_i^{i \in 1..j-1}, l_j = v_j, l_k = t_k^{k \in j+1..n}\} \longrightarrow \{l_i = v_i^{i \in 1..j}, l_k = t_k^{k \in j+1..n}\} \quad (\text{E-RCDV})$	$\frac{t_0 \longrightarrow t'_0}{\text{case } t_0 \text{ of } C_i \ x_i \Rightarrow t_i^{i \in 1..n} \longrightarrow \text{case } t'_0 \text{ of } C_i \ x_i \Rightarrow t_i^{i \in 1..n}}$
$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{\$t_1 \mid \mu \longrightarrow \$t'_1 \mid \mu'} \quad (\text{E-DEREF})$	$\text{case } C_j \ v_j \text{ of } C_i \ x_i \Rightarrow t_i^{i \in 1..n} \longrightarrow [x_j \mapsto v_j] \ t_j \quad (\text{E-CASE-TY})$
$\frac{\mu(l) = v}{\$l \mid \mu \longrightarrow v \mid \mu} \quad (\text{E-DEREFLOC})$	$\frac{v_1 \equiv v_{1,i}^{i \in 0..n-1}}{v_1 [v_2] \longrightarrow v_{1,v_2}} \quad (\text{E-ARR-ACC})$
$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{t_1 := t_2 \mid \mu \longrightarrow t'_1 := t_2 \mid \mu'} \quad (\text{E-ASSIGN1})$	$\frac{t_2 \longrightarrow t'_2}{v_1 [t_2] \longrightarrow v_1 [t'_2]} \quad (\text{E-ARR-ACC1})$
$\frac{t_2 \mid \mu \longrightarrow t'_2 \mid \mu'}{l := t_2 \mid \mu \longrightarrow l := t'_2 \mid \mu'} \quad (\text{E-ASSIGN2})$	$\frac{t_1 \mid \sigma \longrightarrow t'_1 \mid \sigma'}{\text{sw } t_1 \mid \sigma \longrightarrow \text{sw } t'_1 \mid \sigma'} \quad (\text{E-SW})$
$l := v_1 \mid \mu \longrightarrow \text{unit} \mid [l \mapsto v_1] \ \mu \quad (\text{E-ASSIGN})$	$\frac{w \notin \text{dom}(\sigma)}{\text{sw } v_1 \mid \sigma \longrightarrow w \mid (\sigma, w \mapsto v_1)} \quad (\text{E-SWV})$
$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{\text{ref } t_1 \mid \mu \longrightarrow \text{ref } t'_1 \mid \mu'} \quad (\text{E-REF})$	$\frac{t_1 \mid \sigma \longrightarrow t'_1 \mid \sigma'}{\text{hw } t_1 \mid \sigma \longrightarrow \text{hw } t'_1 \mid \sigma'} \quad (\text{E-HW})$
	$\frac{\sigma(w) = v}{\text{hw } w \mid \sigma \longrightarrow v \mid \sigma} \quad (\text{E-HWWRAP})$

4.2 Proofs

In this section, we will prove certain characteristics and properties of the language.

Definition 1 (Normal Form). *A term t is in normal form if no evaluation rule applies to it.*

Definition 2 (Stuck State). *A closed term is stuck if it is in normal form but is not a value.*

Definition 3 (Typing Relation). *The typing relation is the smallest binary relation between terms and types satisfying all instances of the typing rules from §4.1.5.*

Lemma 1 (Inversion of the Typing Relation). *The following are true.*

1. If $\Gamma \vdash x : R$, then $x : R \in \Gamma$.
2. If $\Gamma \vdash \lambda x : T_1. t_2 : R$, then $R = T_1 \rightarrow R_2$ for some R_2 with $\Gamma, x : T_1 \vdash t_2 : R_2$.
3. If $\Gamma \vdash t_1 t_2 : R$ then there is some type T_{11} such that $\Gamma \vdash t_1 : T_{11} \rightarrow R$ and that $\Gamma \vdash t_2 : T_{11}$.
4. If $\langle \text{integer} \rangle : R$, then $R = \text{int}$.
5. If $\langle \text{real} \rangle : R$, then $R = \text{real}$.
6. If $\langle \text{string} \rangle : R$, then $R = \text{string}$.
7. If $\text{'b':0} : R$, then $R = \text{bit}$.
8. If $\text{'b':1} : R$, then $R = \text{bit}$.
9. If $\text{nil} : R$, then $R = T_S \text{ list}$.
10. If $() : R$, then $R = \text{unit}$.
11. If $\sim t_1 : \text{int}$, then $t_1 : \text{int}$.
12. If $t_1 + t_2 : R$, then $R = \text{int}$, $t_1 : \text{int}$ and $t_2 : \text{int}$.
13. If $t_1 - t_2 : R$, then $R = \text{int}$, $t_1 : \text{int}$ and $t_2 : \text{int}$.
14. If $t_1 * t_2 : R$, then $R = \text{int}$, $t_1 : \text{int}$ and $t_2 : \text{int}$.
15. If $t_1 / t_2 : R$, then $R = \text{int}$, $t_1 : \text{int}$ and $t_2 : \text{int}$.
16. If $t_1 \% t_2 : R$, then $R = \text{int}$, $t_1 : \text{int}$ and $t_2 : \text{int}$.
17. If $\sim t_1 : \text{real}$, then $t_1 : \text{real}$.
18. If $t_1 +. t_2 : R$, then $R = \text{real}$, $t_1 : \text{real}$ and $t_2 : \text{real}$.
19. If $t_1 -. t_2 : R$, then $R = \text{real}$, $t_1 : \text{real}$ and $t_2 : \text{real}$.

20. If $t_1 *. t_2 : R$, then $R = \text{real}$, $t_1 : \text{real}$ and $t_2 : \text{real}$.
21. If $t_1 /. t_2 : R$, then $R = \text{real}$, $t_1 : \text{real}$ and $t_2 : \text{real}$.
22. If $!t_1 : R$, then $R = T_H$ and $t_1 : T_H$.
23. If $t_1 \& t_2 : R$, then $R = T_H$, $t_1 : T_H$ and $t_2 : T_H$.
24. If $t_1 | t_2 : R$, then $R = T_H$, $t_1 : T_H$ and $t_2 : T_H$.
25. If $t_1 \wedge t_2 : R$, then $R = T_H$, $t_1 : T_H$ and $t_2 : T_H$.
26. If $\&\rightarrow t_1 : R$, then $R = \text{bit}$ and $t_1 : \text{bit}[n]$.
27. If $|\rightarrow t_1 : R$, then $R = \text{bit}$ and $t_1 : \text{bit}[n]$.
28. If $\wedge\rightarrow t_1 : R$, then $R = \text{bit}$ and $t_1 : \text{bit}[n]$.
29. If $t_1 \&\& t_2 : R$, then $R = \text{bit}$, $t_1 : \text{bit}[n]$ and $t_2 : \text{bit}[n]$.
30. If $t_1 || t_2 : R$, then $R = \text{bit}$, $t_1 : \text{bit}[n]$ and $t_2 : \text{bit}[n]$.
31. If $t_1 \wedge\wedge t_2 : R$, then $R = \text{bit}$, $t_1 : \text{bit}[n]$ and $t_2 : \text{bit}[n]$.
32. If $t_1 << t_2 : R$, then $R = \text{bit}[n]$, $t_1 : \text{bit}[n]$ and $t_2 : \text{bit}[m]$.
33. If $t_1 >> t_2 : R$, then $R = \text{bit}[n]$, $t_1 : \text{bit}[n]$ and $t_2 : \text{bit}[m]$.
34. If $t_1 >>> t_2 : R$, then $R = \text{bit}[n]$, $t_1 : \text{bit}[n]$ and $t_2 : \text{bit}[m]$.
35. If $t_1 = t_2 : R$, then $R = \text{int}$, $t_1 : T_S$ and $t_2 : T_S$.
36. If $t_1 <> t_2 : R$, then $R = \text{int}$, $t_1 : T_S$ and $t_2 : T_S$.
37. If $t_1 < t_2 : R$, then $R = \text{int}$, $t_1 : T_S$ and $t_2 : T_S$.
38. If $t_1 <= t_2 : R$, then $R = \text{int}$, $t_1 : T_S$ and $t_2 : T_S$.
39. If $t_1 > t_2 : R$, then $R = \text{int}$, $t_1 : T_S$ and $t_2 : T_S$.
40. If $t_1 >= t_2 : R$, then $R = \text{int}$, $t_1 : T_S$ and $t_2 : T_S$.
41. If $t_1 \text{ andalso } t_2 : R$, then $R = \text{int}$, $t_1 : \text{int}$ and $t_2 : \text{int}$.
42. If $t_1 \text{ orelse } t_2 : R$, then $R = \text{int}$, $t_1 : \text{int}$ and $t_2 : \text{int}$.
43. If $\text{not } t_1 : R$, then $R = \text{int}$ and $t_1 : \text{int}$.
44. If $t_1 :: t_2 : R$, then $R = T_S \text{ list}$, $t_1 : T_S$ and $t_2 : T_S \text{ list}$.
45. If $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{int}$, $t_2 : R$, and $t_3 : R$.

46. If `if` t_1 `then` $t_2 : R$, then $R = \text{unit}$, $t_1 : \text{int}$ and $t_2 : \text{unit}$.
47. If `ref` $t_1 : R$, then $R = T_S \text{ ref}$ and $t_1 : T_S$.
48. If $t_1 := t_2 : R$, then $R = \text{unit}$, $t_1 : T_S \text{ ref}$ and $t_2 : T_S$.
49. If $\$t_1 : R$, then $R = T_S$ and $t_1 : T_S \text{ ref}$.
50. If `sw` $t_1 : R$, then $R = T_H \text{ sw}$ and $t_1 : T_H$.
51. If $t_1[t_2] : R$, then $R = T_H$, $t_1 : T_H[n]$ and $t_2 : \text{int}$.
52. If $\{l_i = t_i^{i \in 1..n}\} : R$, then $R = \{l_i : T_i^{i \in 1..n}\}$ and for each i , $t_i : T_i$.
53. If $\#l_j \ t_1 : R$, then $R = T_j$ and $t_1 : \{l_i : T_i^{i \in 1..n}\}$.
54. If `let` $x = t_1$ `in` t_2 `end` : R , then $\Gamma, \tau \vdash R = T_2$, $\Gamma, \tau \vdash t_1 : T_1$ and $\Gamma, \tau, x : T_1 \vdash t_2 : T_2$.
55. If $C_j \ t : R$, then $R = \langle C_i : T_i \rangle^{i \in 1..n}$ and $t : T_j$.
56. If `case` t_0 `of` $C_i \ x_i \Rightarrow t_i^{i \in 1..n} : R$, then $R = T$, $t_0 : \langle C_i : T_i \rangle^{i \in 1..n}$, and for each i , $\Gamma, x_i : T_i \vdash t_i : T$.

Proof. Immediate from the definition of the typing relation. \square

Lemma 2 (Canonical Forms). *The following are true.*

1. If v is a value of type `int`, then v is an integer value according to the grammar in §4.1.1.1.
2. If v is a value of type `real`, then v is a real value according to the grammar in §4.1.1.1.
3. If v is a value of type `string`, then v is a string according to the grammar in §4.1.1.1.
4. If v is a value of type `bit`, then v is either `'b:0` or `'b:1`.
5. If v is a value of type $T_1 \rightarrow T_2$, then $v = \lambda x : T_1. t_2$.
6. If v is a value of type $T_S \text{ ref}$, then v is a location in store μ .
7. If v is a value of type $\{l_i : T_i^{i \in 1..n}\}$, then v is a value with form $\{l_i = v_i^{i \in 1..n}\}$.
8. If C is a constructor of datatype D accepting type T_1 and v is a value of type T_1 then $C \ v$ is a value of type D with form $\langle C = v \rangle$.

Proof. We refer to the first 10 cases of the inversion lemma as they pertain to values in this language and to the grammars in §4.1.1.1 and §4.1.2.1.

Case (1). Values in this language can take several forms. The case of an `<integer>` gives us our desired result immediately. All other forms cannot occur since we assumed that v has type `int` and among the first 10 cases of the inversion lemma, only case 4 tells us that the value has type `int`.

The remaining cases are similar. \square

Theorem 3 (Progress). *Suppose t is a closed, well-typed term ($\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \rightarrow t'$.*

Proof. By induction on a derivation of $t : T$.

Case T-INT, T-REAL, T-STRING, T-BIT, T-NIL:

Immediate since t is a value.

Case T-VAR:

Cannot occur since t must be closed as per the hypothesis.

Case T-ABS:

Immediate since t is a value.

Case T-APP:

$$\begin{aligned} t &= t_1 \ t_2 \\ \vdash t_1 : T_{11} \rightarrow T_{12} \\ \vdash t_2 : T_{11} \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 for which $t_1 \rightarrow t'_1$, and likewise for t_2 . If $t_1 \rightarrow t'_1$, then by E-APP1, $t \rightarrow t'_1 \ t_2$. On the other hand, if t_1 is a value and $t_2 \rightarrow t'_2$, then by E-APP2, $t \rightarrow t_1 \ t'_2$. Finally, if both t_1 and t_2 are values, then case 5 of the canonical forms lemma tells us that t_1 has the form $\lambda x : T_{11}. t_{12}$, and so by E-APPABS, $t \rightarrow [x \mapsto t_2] t_{12}$.

Case T-INT-NEG:

$$\begin{aligned} t &= \sim t_1 \\ t_1 &: \text{int} \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If t_1 is a value, then case 1 of the canonical forms lemma assures us that it is an integer value as described in §4.1.1.1 with existing and valid semantic meaning for negation, yielding a value.

On the other hand, if $t_1 \rightarrow t'_1$, then by E-NEG, $t \rightarrow \sim t'_1$.

Case T-INT-ADD:

$$\begin{aligned} t &= t_1 + t_2 \\ t_1 &: \text{int} \\ t_2 &: \text{int} \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$, then by E-INT-ADD1, $t \rightarrow t'_1 + t_2$. On the other hand, if t_1 is a value, then case 1 of the canonical forms lemma assures us that it is an integer value as described in §4.1.1.1. Further in this case, by the induction hypothesis, either t_2 is a value or else there is some t'_2 such that $t_2 \rightarrow t'_2$. If $t_2 \rightarrow t'_2$, then by E-INT-ADD2, $t \rightarrow v_1 + t'_2$. On the other hand, if t_2 is a value, then case 1 of the canonical forms lemma assures us that it is an integer value as described in §4.1.1.1. Thus, both t_1 and t_2 are integer values in this case and addition has a well-defined semantic meaning thereby yielding a value.

Case T-INT-(SUB/MUL/DIV/MOD):

Similar to T-INT-ADD.

Case T-REAL-NEG:

$$\begin{aligned} t &= \sim t_1 \\ t_1 &: \text{real} \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If t_1 is a value, then case 2 of the canonical forms lemma assures us that it is a value in the domain of real numbers as described in §4.1.1.1 with existing and valid semantic meaning for negation, yielding a value. On the other hand, if $t_1 \rightarrow t'_1$, then by E-NEG, $t \rightarrow \sim t'_1$.

Case T-REAL-ADD:

$$\begin{aligned} t &= t_1 +. t_2 \\ t_1 &: \text{real} \\ t_2 &: \text{real} \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$, then by E-REAL-ADD1, $t \rightarrow t'_1 +. t_2$. On the other hand, if t_1 is a value, then case 2 of the canonical forms lemma assures us that it is a value in the domain of real numbers as described in §4.1.1.1. Further in this case, by the induction hypothesis, either t_2 is a value or else there is some t'_2 such that $t_2 \rightarrow t'_2$. If $t_2 \rightarrow t'_2$, then by E-REAL-ADD2, $t \rightarrow v_1$

$+ t'_2$. On the other hand, if t_2 is a value, then case 2 of the canonical forms lemma assures us that it is a value in the domain of real numbers as described in §4.1.1.1. Thus, both t_1 and t_2 are real values in this case and addition has a well-defined semantic meaning thereby yielding a value.

Case T-REAL-(SUB/MUL/DIV):

Similar to T-REAL-ADD.

Case T-BIT-NEG:

$$\begin{aligned} t &= !t_1 \\ t_1 &: T_H \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$ then by E-BIT-NEG1, $t \rightarrow !t'_1$. On the other hand, if t_1 is a value then it is a hardware value and we can perform structural induction on the possible types. If it is a bit then by E-BIT-NEG we produce a logical not-gate. If it is a hardware array then by E-BIT-NEG2 we move the negation inside and apply to each element. Similarly, if it is a hardware record then by E-BIT-NEG3 we move the negation inside and apply to each field element.

Case T-AND:

$$\begin{aligned} t &= t_1 \& t_2 \\ t_1 &: T_H \\ t_2 &: T_H \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$, then by E-AND1, $t \rightarrow t'_1 \& t_2$. On the other hand, if t_1 is a value, then it is a hardware value. Further in this case, by the induction hypothesis, either t_2 is a value or else there is some t'_2 such that $t_2 \rightarrow t'_2$. If $t_2 \rightarrow t'_2$, then by E-AND2, $t \rightarrow v_1 \& t'_2$. On the other hand, if t_2 is a value, then it is a hardware value. If, both t_1 and t_2 are hardware values then we can perform structural induction on the possible types. If $T_H = \text{bit}$ then by the derived-term definition, $t \rightarrow \&->\#[t_1, t_2]$. If it is a hardware array then by E-AND3 we move the operation inside and apply to each pair of elements. Similarly, if it is a hardware record then by E-AND4 we move the operation inside and apply to each pair of field elements.

Case T-(OR/XOR):

Similar to T-AND.

Case T-AND-RED:

$t = \&->t_1$
 $t_1 : \text{bit}[n]$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$ then by E-AND-RED1, $t \rightarrow \&->t'_1$. On the other hand if t_1 is a value, then it is a bit array and is evaluated by E-AND-RED.

Case T-(OR/XOR)-RED:

Similar to T-AND-RED.

Case T-SLL:

$t_1 \ll t_2$
 $t_1 : \text{bit}[n]$
 $t_2 : \text{bit}[m]$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$ then by E-SLL1, $t \rightarrow t'_1 \ll t_2$. On the other hand, if t_1 is a value, then either t_2 is a value or else there is some t'_2 such that $t_2 \rightarrow t'_2$. If $t_2 \rightarrow t'_2$ then by E-SLL2, $t \rightarrow v_1 \ll t'_2$. On the other hand, if t_2 is a value, then both t_1 and t_2 are values and by the definition in §4.1.2.1, t is a value.

Case T-(SRL/SRA):

Similar to T-SLL.

Case T-EQ:

$t = t_1 = t_2$
 $t_1 : T_S$
 $t_2 : T_S$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$, then by E-EQ1, $t \rightarrow t'_1 = t_2$. On the other hand, if t_1 is a value, then it may be of any software type. Further in this case, by the induction hypothesis, either t_2 is a value or else there is some t'_2 such that $t_2 \rightarrow t'_2$. If $t_2 \rightarrow t'_2$, then by E-EQ2, $t \rightarrow v_1 = t'_2$. On the other hand, if t_2 is a value, then it may be of any software type. Further, both t_1 and t_2 are values in this case and equality has semantic meaning defined as per the type definition, thereby yielding a value.

Case T-(NEQ/LT/LEQ/GT/GEQ):

Similar to T-EQ.

Case T-NOT:

$t = \text{not } t_1$
 $t_1 : \text{int}$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. In either case, t can be evaluated by E-NOT, namely $t \rightarrow \text{if } t_1 \text{ then } 0 \text{ else } 1$.

Case T-CONS:

$t = t_1 :: t_2$
 $t_1 : T_S$
 $t_2 : T_S \text{ list}$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$, then by E-CONS1, $t \rightarrow t'_1 :: t_2$. On the other hand, if t_1 is a value, then either t_2 is a value or else there is some t'_2 such that $t_2 \rightarrow t'_2$. If $t_2 \rightarrow t'_2$, then by E-CONS2, $t \rightarrow v_1 :: t'_2$. On the other hand, if t_2 is a value, then both t_1 and t_2 are values and list concatenation evaluates under well-defined semantics yielding a value.

Case T-IFELSE:

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$
 $t_1 : \text{int}$
 $t_2 : T$
 $t_3 : T$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$, then by E-IFELSE, $t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$. On the other hand, if t_1 is a value, then case 1 of the canonical forms lemma assures us that it is an integer value as described in §4.1.1.1. Further, it is either zero or non-zero, and can be evaluated by either E-IFELSE-T or E-IFELSE-F.

Case T-REF:

$t = \text{ref } t_1$
 $t_1 : T_S$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$, then by E-REF, $t \rightarrow \text{ref } t'_1$. On the other hand, if t_1 is a value, then by E-REFV, $t \rightarrow l | (\mu, l \mapsto v_1)$.

Case T-SW-H:

$t = \text{sw } t_1$
 $t_1 : T_H$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$ then by E-SW, $t \rightarrow \text{sw } t'_1$. On the other hand, if t_1 is a value then by E-SWV, $t \rightarrow w | (\sigma, w \mapsto v_1)$.

Case T-SW-M:

Similar to T-SW-H.

Case T-HW-H:

$$\begin{aligned} t &= \text{hw } t_1 \\ t_1 &: T_H \text{ sw} \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$ then by E-HW, $t \rightarrow \text{hw } t'_1$. On the other hand, if t_1 is a value then by E-HWWRAP, $t \rightarrow w | \sigma$.

Case T-HW-M:

Similar to T-HW-H.

Case T-ASSIGN:

$$\begin{aligned} t &= t_1 := t_2 \mid \mu \\ t_1 &: T_S \text{ ref} \\ t_2 &: T_S \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \mid \mu \rightarrow t'_1 \mid \mu'$. If $t_1 \mid \mu \rightarrow t'_1 \mid \mu'$, then by E-ASSIGN1, $t \rightarrow t'_1 := t_2 \mid \mu'$. On the other hand, if t_1 is a value, then by the induction hypothesis, either t_2 is a value or else there is some t'_2 such that $t_2 \mid \mu \rightarrow t'_2 \mid \mu'$. If $t_2 \mid \mu \rightarrow t'_2 \mid \mu'$, then by E-ASSIGN2, $t \rightarrow v_1 := t'_2 \mid \mu'$. On the other hand, if t_2 is a value, then both t_1 and t_2 are values and so by E-ASSIGN, $t \rightarrow \text{unit} \mid [l \mapsto v_1] \mu$.

Case T-DEREF:

$$\begin{aligned} t &= \$t_1 \mid \mu \\ t_1 &: T_S \text{ ref} \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 and μ' such that $t_1 \mid \mu \rightarrow t'_1 \mid \mu'$. If $t_1 \mid \mu \rightarrow t'_1 \mid \mu'$, then by E-DEREF, $t \rightarrow \$t'_1 \mid \mu'$. On the other hand, if t_1 is a value then case 6 of the canonical forms lemma assures us that it is a location l in store μ , and so assuming that $\mu(l) = v$ then by E-DEREFLOC, $t \rightarrow v \mid \mu$.

Case T-ARR-ACC:

$$\begin{aligned} t &= t_1[t_2] \\ t_1 &: T_H[n] \\ t_2 &: \text{int} \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. Since t_1 is of hardware type it is a value by definition. By the induction hypothesis, either t_2 is a value or else there is some t'_2 such that $t_2 \rightarrow t'_2$. If $t_2 \rightarrow t'_2$, then by E-ARR-ACC1, $t \rightarrow t_1[t'_2]$. On the other hand, if t_2 is a value, then both t_1 and t_2 are values and by E-ARR-ACC, $t \rightarrow v_{1,v_2}$.

Case T-RCD:

$$\begin{aligned} t &= \{l_i = t_i^{i \in 1..n}\} \\ \text{for each } i, t_i &: T_i \end{aligned}$$

By the induction hypothesis, for all j either t_j is a value or else there is some t'_j such that $t_j \rightarrow t'_j$. If $t_j \rightarrow t'_j$, then by E-RCD, $t \rightarrow \{l_i = v_i^{i \in 1..j-1}, l_j = t'_j, l_k = t_k^{k \in j+1..n}\}$. On the other hand, if t_j is a value, then by E-RCDV, $t \rightarrow \{l_i = v_i^{i \in 1..j}, l_k = t_k^{k \in j+1..n}\}$.

Case T-PROJ:

$$\begin{aligned} t &= \#l_j t_1 \\ t_1 &: \{l_i : T_i^{i \in 1..n}\} \end{aligned}$$

By the induction hypothesis, either t_1 is a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$ then by E-PROJ, $t \rightarrow \#l_j t'_1$. On the other hand, if t_1 is a value then case 7 of the canonical forms lemma assures us that it is a record value of form $\{l_i = v_i^{i \in 1..n}\}$ and so by E-PROJ-RCD, $t \rightarrow v_j$.

Case T-LET:

$$\begin{aligned} t &= \text{let } x_1 = t_1(x_i = t_i)^{i \in 2..n} \text{ in} \\ & t_0 \text{ end } t_1 : T_1 \end{aligned}$$

By the induction hypothesis, t_1 is either a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$, then by E-LET, $t \rightarrow \text{let } x_1 = t'_1(x_i = t_i)^{i \in 2..n} \text{ in } t_0 \text{ end}$. On the other hand, if t_1 is a value then by the definition of let-bindings $n \geq 1$. If $n = 1$, then by E-LETV2, $t \rightarrow [x_1 \mapsto v_1] t_0$. On the other hand, if $n > 1$, then by E-LETV1, $t \rightarrow \text{let } x_2 = t_2(x_i = t_i)^{i \in 3..n} \text{ in } t_0 \text{ end}$.

Case T-DATATY:

$$\begin{aligned} t &= C_j t_1 \\ t_1 &: T_j \end{aligned}$$

By the induction hypothesis, t_1 is either a value or else there is some t'_1 such that $t_1 \rightarrow t'_1$. If $t_1 \rightarrow t'_1$, then by E-DATATY, $t \rightarrow C_j t'_1$. On the other hand, if t_1 is a value then case 8 of the canonical forms lemma assures us that it is a datatype value of form $\langle C_j = t_1 \rangle$.

Case T-CASE:

$$t = \text{case } t_0 \text{ of } C_i \ x_i \Rightarrow t_i^{i \in 1..n} \\ t_0 : \langle C_i : T_i \rangle^{i \in 1..n}$$

By the induction hypothesis, t_0 is either a value or else there is some t'_0 such that $t_0 \rightarrow t'_0$. If $t_0 \rightarrow t'_0$, then by E-CASE, $t \rightarrow \text{case } t'_0 \text{ of } C_i \ x_i \Rightarrow t_i^{i \in 1..n}$. On the other hand, if t_0 is a value then case 8 of the canonical forms lemma assures us that it is a datatype value of form $\langle C_j = v_j \rangle$ and so by E-CASE-TY, $t \rightarrow [x_j \mapsto v_j] t_j$.

□

Theorem 4 (Preservation). *If $t : T$ and $t \rightarrow t'$, then $t' : T$.*

Proof. By induction on a derivation of $t : T$. At each step of the induction, we assume that the desired property holds for all subderivations (i.e. that if $s : S$ and $s \rightarrow s'$, then $s' : S$, whenever $s : S$ is proved by a subderivation of the present one) and proceed by case analysis on the final rule in the derivation.

Case T-VAR:

$$t = x \\ x : T \in \Gamma$$

If the last rule in the derivation is T-VAR, then we know from the form of this rule that t must be a variable of type T . Thus, t is a value, so it cannot be the case that $t \rightarrow t'$ for any t' , and the requirements of the theorem are vacuously satisfied.

Case T-ABS:

$$t = \lambda x : T_1. t_2$$

If the last rule in the derivation is T-ABS, then we know from the form of this rule that t must be an abstraction. Thus, t is a value, so it cannot be the case that $t \rightarrow t'$ for any t' , and the requirements of the theorem are vacuously satisfied.

Case T-APP:

$$t = t_1 \ t_2 \\ \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \\ \Gamma \vdash t_2 : T_{11} \\ T = T_{12}$$

Looking at the evaluation rules with applications on the left-hand side, we find that there are three rules by which $t \rightarrow t'$ can be derived: E-APP1, E-APP2, and E-APPABS. We consider each case separately.

Subcase E-APP1:

$$t_1 \rightarrow t'_1 \\ t' = t'_1 \ t_2$$

From the assumptions of the T-APP case, we have a subderivation of the original typing derivation whose conclusion is $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$. We can apply the induction hypothesis to this subderivation obtaining $\Gamma \vdash t'_1 : T_{11} \rightarrow T_{12}$. Combining this with the fact that $\Gamma \vdash t_2 : T_{11}$, we can apply rule T-APP to conclude that $\Gamma \vdash t' : T$.

Subcase E-APP2:

Similar to E-APP1.

Subcase E-APPABS:

$$t_1 = \lambda x : T_{11}. t_{12} \ t_2 = v_2 \ t' \\ = [x \mapsto v_2] t_{12}$$

Using the inversion lemma, we can deconstruct the typing derivation for $\lambda x : T_{11}. t_{12}$ yielding $\Gamma, x : T_{11} \vdash t_{12} : T_{12}$. From this we obtain $\Gamma \vdash t' : T_{12}$.

Case T-INT:

$$t = \langle \text{integer} \rangle \\ T = \text{int}$$

If the last rule in the derivation is T-INT, then we know from the form of this rule that t must be an integer value as described in §4.1.1.1 and that T is `int`. Thus, t is a value, so it cannot be the case that $t \rightarrow t'$ for any t' , and the requirements of the theorem are vacuously satisfied.

Case T-REAL, T-STRING, T-BIT, T-NIL:

Similar to T-INT.

Case T-INT-NEG:

$$t = \sim t_1 \\ T = \text{int}$$

If the last rule in the derivation is T-INT-NEG, then we know from the form of this rule that t must have the form $\sim t_1$ and that T is int . We must further have a subderivation with conclusion $t_1 : \text{int}$. Looking at the evaluation rules with integer negation on the left-hand side, there is one rule by which $t \longrightarrow t'$ can be derived: E-NEG.

Subcase E-NEG:

$$\begin{aligned} t_1 &\longrightarrow t'_1 \\ t' &= \sim t'_1 \end{aligned}$$

From the assumptions of the T-INT-NEG case, we have a subderivation of the original typing derivation with conclusion $t_1 : \text{int}$. We may apply the induction hypothesis to this subderivation, obtaining $t'_1 : \text{int}$. Then, by T-INT-NEG, $\sim t'_1 : \text{int}$ and so $t' : \text{int}$.

Case T-INT-ADD:

$$\begin{aligned} t &= t_1 + t_2 \\ T &= \text{int} \end{aligned}$$

If the last rule in the derivation is T-INT-ADD, then we know from the form of this rule that t must have the form $t_1 + t_2$ and that T is int . We must further have subderivations with conclusions $t_1 : \text{int}$ and $t_2 : \text{int}$. Looking at the evaluation rules with integer addition on the left-hand side, there are two rules by which $t \longrightarrow t'$ can be derived: E-INT-ADD1 and E-INT-ADD2.

Subcase E-INT-ADD1:

$$\begin{aligned} t_1 &\longrightarrow t'_1 \\ t' &= t'_1 + t_2 \end{aligned}$$

From the assumptions of the T-INT-ADD case, we have a subderivation of the original typing derivation with conclusion $t_1 : \text{int}$. We may apply the induction hypothesis to this subderivation, obtaining $t'_1 : \text{int}$. We also have a subderivation with conclusion $t_2 : \text{int}$. Thus, by T-INT-ADD, $t'_1 + t_2 : \text{int}$ and so $t' : \text{int}$.

Subcase E-INT-ADD2:

$$\begin{aligned} t_2 &\longrightarrow t'_2 \\ t' &= v_1 + t'_2 \end{aligned}$$

From the assumptions of the T-INT-ADD case, we have a subderivation of the original typing derivation with conclusion $t_2 : \text{int}$. We may apply

the induction hypothesis to this subderivation, obtaining $t'_2 : \text{int}$. We also have a subderivation with conclusion $t_1 : \text{int}$ and since v_1 is the value form denotation of t_1 , it also holds that $v_1 : \text{int}$. Thus, by T-INT-ADD, $v_1 + t'_2 : \text{int}$ and so $t' : \text{int}$.

Case T-INT-(SUB/MUL/DIV/MOD):

Similar to T-INT-ADD.

Case T-REAL-(ADD/SUB/MUL/DIV):

Similar to T-INT-ADD.

Case T-BIT-NEG:

$$\begin{aligned} t &= !t_1 \\ T &= T_H \end{aligned}$$

If the last rule in the derivation is T-BIT-NEG, then we know from the form of this rule that t must have the form $!t_1$ and that T is T_H . Looking at the evaluation rules there is only one rule by which $t \longrightarrow t'$ can be derived: E-BIT-NEG1. We may apply the induction hypothesis to our subderivation, obtaining $t'_1 : T_H$. Then, by T-BIT-NEG, $!t'_1 : T_H$ and so $t' : T_H$.

Case T-AND:

$$\begin{aligned} t &= t_1 \ \& \ t_2 \\ T &= T_H \end{aligned}$$

If the last rule in the derivation is T-AND, then we know from the form of this rule that t must have the form $t_1 \ \& \ t_2$ and that T is T_H . We must further have subderivations with conclusions $t_1 : T_H$ and $t_2 : T_H$. Looking at the evaluation rules with $\&$ on the left-hand side, there are two rules by which $t \longrightarrow t'$ can be derived: E-AND1 and E-AND2.

Subcase E-AND1:

$$\begin{aligned} t_1 &\longrightarrow t'_1 \\ t' &= t'_1 \ \& \ t_2 \end{aligned}$$

From the assumptions of the T-AND case, we have a subderivation of the original typing derivation with conclusion $t_1 : T_H$. We may apply the induction hypothesis to this subderivation, obtaining $t'_1 : T_H$. We also have a subderivation with conclusion $t_2 : T_H$. Thus, by T-AND, $t'_1 \ \& \ t_2 : T_H$ and so $t' : T_H$.

Subcase E-AND2:

$$\begin{aligned} t_2 &\longrightarrow t'_2 \\ t' &= v_1 \ \& \ t'_2 \end{aligned}$$

From the assumptions of the T-AND case, we have a subderivation of the original typing derivation with conclusion $t_2 : T_H$. We may apply the induction hypothesis to this subderivation, obtaining $t'_2 : T_H$. We also have a subderivation with conclusion $t_1 : T_H$ and since v_1 is the value form denotation of t_1 , it also holds that $v_1 : T_H$. Thus, by T-INT-ADD, $v_1 \& t'_2 : T_H$ and so $t' : T_H$.

Case T-(OR/XOR):

Similar to T-AND.

Case T-AND-RED:

$$\begin{aligned} t &= \&->t_1 \\ T &= \text{bit}[n] \end{aligned}$$

If the last rule in the derivation is T-AND-RED, then we know from the form of this rule that t must have the form $\&->t_1$ and that T is $\text{bit}[n]$. Looking at the evaluation rules there is only one rule by which $t \longrightarrow t'$ can be derived: E-AND-RED1. We may apply the induction hypothesis to our subderivation, obtaining $t'_1 : T_H$. Then, by T-AND-RED, $\&->t'_1 : \text{bit}[n]$ and so $t' : \text{bit}[n]$.

Case T-(OR/XOR)-RED:

Similar to T-AND-RED.

Case T-SLL:

$$\begin{aligned} t &= t_1 \ll t_2 \\ T &= \text{bit}[n] \end{aligned}$$

If the last rule in the derivation is T-SLL, then we know from the form of this rule that t must have the form $t_1 \ll t_2$ and that T is $\text{bit}[n]$. We must further have subderivations with conclusions $t_1 : \text{bit}[n]$ and $t_2 : \text{bit}[m]$. Looking at the evaluation rules with left-shifting on the left-hand side, there are two rules by which $t \longrightarrow t'$ can be derived: E-SLL1 and E-SLL2.

Subcase E-SLL1:

$$\begin{aligned} t_1 &\longrightarrow t'_1 \\ t' &= t'_1 \ll t_2 \end{aligned}$$

From the assumptions of the T-SLL case, we have a subderivation of the original typing derivation with conclusion $t_1 : \text{bit}[n]$. We may apply the induction hypothesis to this subderivation, obtaining $t'_1 : \text{bit}[n]$. We

also have a subderivation with conclusion $t_2 : \text{bit}[m]$. Thus, by T-SLL, $t'_1 \ll t_2 : \text{bit}[n]$ and so $t' : \text{bit}[n]$.

Subcase E-SLL2:

$$\begin{aligned} t_2 &\longrightarrow t'_2 \\ t' &= v_1 \ll t'_2 \end{aligned}$$

From the assumptions of the T-SLL case, we have a subderivation of the original typing derivation with conclusion $t_2 : \text{bit}[m]$. We may apply the induction hypothesis to this subderivation, obtaining $t'_2 : \text{bit}[m]$. We also have a subderivation with conclusion $t_1 : \text{bit}[n]$ and since v_1 is the value form denotation of t_1 , it also holds that $v_1 : \text{bit}[n]$. Thus, by T-SLL, $v_1 \ll t'_2 : \text{bit}[n]$ and so $t' : \text{bit}[n]$.

Case T-(SRL/SRA):

Similar to T-SLL.

Case T-EQ:

$$\begin{aligned} t &= t_1 = t_2 \\ T &= \text{int} \end{aligned}$$

If the last rule in the derivation is T-EQ, then we know from the form of this rule that t must have the form $t_1 = t_2$ and that T is int . We must further have subderivations with conclusions $t_1 : T_S$ and $t_2 : T_S$ for some T_S . Looking at the evaluation rules with equality on the left-hand side, there are two rules by which $t \longrightarrow t'$ can be derived: E-EQ1 and E-EQ2.

Subcase E-EQ1:

$$\begin{aligned} t_1 &\longrightarrow t'_1 \\ t' &= t'_1 = t_2 \end{aligned}$$

From the assumptions of the T-EQ case, we have a subderivation of the original typing derivation with conclusion $t_1 : T_S$. We may apply the induction hypothesis to this subderivation, obtaining $t'_1 : T_S$. We also have a subderivation with conclusion $t_2 : T_S$. Thus, by T-EQ, $t'_1 = t_2 : \text{int}$ and so $t' : \text{int}$.

Subcase E-INT-ADD2:

$$\begin{aligned} t_2 &\longrightarrow t'_2 \\ t' &= v_1 = t'_2 \end{aligned}$$

From the assumptions of the T-EQ case, we have a subderivation of the original typing derivation with conclusion $t_2 : T_S$. We may apply the induction hypothesis to this subderivation,

obtaining $t'_2 : T_S$. We also have a subderivation with conclusion $t_1 : T_S$ and since v_1 is the value form denotation of t_1 , it also holds that $v_1 : T_S$. Thus, by T-EQ, $v_1 = t'_2 : \text{int}$ and so $t' : \text{int}$.

Case T-(NEQ/LT/LEQ/GT/GEQ):

Similar to T-EQ.

Case T-CONS:

$t = t_1 :: t_2$
 $T = T_S \text{ list}$

If the last rule in the derivation is T-CONS, then we know from the form of this rule that t must have the form $t_1 :: t_2$ and that T is $T_S \text{ list}$. We must further have subderivations with conclusions $t_1 : T_S$ and $t_2 : T_S \text{ list}$ for some T_S . Looking at the evaluation rules with the **cons** operator on the left-hand side, there are two rules by which $t \longrightarrow t'$ can be derived: E-CONS1 and E-CONS2.

Subcase E-CONS1:

$t_1 \longrightarrow t'_1$
 $t' = t'_1 :: t_2$

From the assumptions of the T-CONS case, we have a subderivation of the original typing derivation with conclusion $t_1 : T_S$. We may apply the induction hypothesis to this subderivation, obtaining $t'_1 : T_S$. We also have a subderivation with conclusion $t_2 : T_S \text{ list}$. Thus, by T-CONS, $t'_1 :: t_2 : T_S \text{ list}$ and so $t' : T_S \text{ list}$.

Subcase E-CONS2:

$t_2 \longrightarrow t'_2$
 $t' = v_1 :: t'_2$

From the assumptions of the T-CONS case, we have a subderivation of the original typing derivation with conclusion $t_2 : T_S \text{ list}$. We may apply the induction hypothesis to this subderivation, obtaining $t'_2 : T_S \text{ list}$. We also have a subderivation with conclusion $t_1 : T_S$ and since v_1 is the value form denotation of t_1 , it also holds that $v_1 : T_S$. Thus, by T-CONS, $v_1 :: t'_2 : T_S \text{ list}$ and so $t' : T_S \text{ list}$.

Case T-IFELSE:

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$
 $T = T_0$

If the last rule in the derivation is T-IFELSE, then we know from the form of this rule that t must have the form **if** t_1 **then** t_2 **else** t_3 and that T is T_0 for some T_0 . We must further have subderivations with conclusions $t_1 : \text{int}$, $t_2 : T_0$ and $t_3 : T_0$. Looking at the evaluation rules with a conditional on the left-hand side, there are three rules by which $t \longrightarrow t'$ can be derived: E-IFELSE, E-IFELSE-T, and E-IFELSE-F.

Subcase E-IFELSE:

$t_1 \longrightarrow t'_1$
 $t' = \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$

From the assumptions of the T-IFELSE case, we have a subderivation of the original typing derivation with conclusion $t_1 : \text{int}$. We may apply the induction hypothesis to this subderivation, obtaining $t'_1 : \text{int}$. We also have subderivations with conclusions $t_2 : T_0$ and $t_3 : T_0$. Thus, by T-IFELSE, **if** t'_1 **then** t_2 **else** $t_3 : T_0$ and so $t' : T_0$.

Subcase E-IFELSE-T:

$t_1 \neq 0$
 $t' = t_2$

If $t \longrightarrow t'$ is derived using E-IFELSE-T, then from the form of this rule we see that $t_1 \neq 0$ and the resulting term t' is the second subexpression t_2 . This means we are finished, since we know (by the assumptions of the T-IF case) that $t_2 : T_0$, which is what we need.

Subcase E-IFELSE-F:

Similar to E-IFELSE-T.

Case T-REF:

$t = \text{ref } t_1 \mid \mu$
 $T = T_S \text{ ref}$

If the last rule in the derivation is T-REF, then we know from the form of this rule that t must have the form **ref** t_1 and that T is $T_S \text{ ref}$ for some T_S . We must further have a subderivation with conclusion $t_1 : T_S$. Looking at the evaluation rules with **ref** on the left-hand side, there are two rules by which $t \longrightarrow t'$ can be derived: E-REF, and E-REFV.

Subcase E-REF:

$t_1 \mid \mu \longrightarrow t'_1 \mid \mu'$
 $t' = \text{ref } t'_1 \mid \mu'$

From the assumptions of the T-REF case, we have a subderivation of the original typing derivation with conclusion $\mathbf{t}_1 : \mathbf{T}_S$. We may apply the induction hypothesis to this subderivation, obtaining $\mathbf{t}'_1 : \mathbf{T}_S$. Thus, by T-REF, $\mathbf{ref} \ \mathbf{t}'_1 \mid \mu' : \mathbf{T}_S \ \mathbf{ref}$ and so $\mathbf{t}' : \mathbf{T}_S \ \mathbf{ref}$.

Subcase E-REFV:

$$\begin{aligned} \mathbf{t}_1 &= \mathbf{v}_1 \\ \mathbf{t}' &= l \mid (\mu, l \mapsto \mathbf{v}_1) \end{aligned}$$

If $\mathbf{t} \longrightarrow \mathbf{t}'$ is derived using E-REFV, then from the form of this rule we see that $\mathbf{t}_1 = \mathbf{v}_1$ and the resulting term \mathbf{t}' is a location in the store μ augmented with a mapping between the location and \mathbf{v}_1 . We know that locations in store μ are of type $\mathbf{T} \ \mathbf{ref}$ where \mathbf{T} is the type of the value to which it is mapped. In this case, that means that the location l is of type $\mathbf{T}_S \ \mathbf{ref}$ which is what we need.

Case T-ASSIGN:

$$\begin{aligned} \mathbf{t} &= \mathbf{t}_1 := \mathbf{t}_2 \\ \mathbf{T} &= \mathbf{unit} \end{aligned}$$

If the last rule in the derivation is T-ASSIGN, then we know from the form of this rule that \mathbf{t} must have the form $\mathbf{t}_1 := \mathbf{t}_2$ and that \mathbf{T} is \mathbf{unit} . We must further have a subderivation with conclusion $\mathbf{t}_1 : \mathbf{T}_S \ \mathbf{ref}$ and $\mathbf{t}_2 : \mathbf{T}_S$ for some \mathbf{T}_S . Looking at the evaluation rules with the assignment operator on the left-hand side, there are three rules by which $\mathbf{t} \longrightarrow \mathbf{t}'$ can be derived: E-ASSIGN1, E-ASSIGN2, and E-ASSIGN.

Subcase E-ASSIGN1:

$$\begin{aligned} \mathbf{t}_1 \mid \mu &\longrightarrow \mathbf{t}'_1 \mid \mu' \\ \mathbf{t}' &= \mathbf{t}'_1 := \mathbf{t}_2 \mid \mu' \end{aligned}$$

From the assumptions of the T-ASSIGN case, we have subderivations of the original typing derivation with conclusions $\mathbf{t}_1 : \mathbf{T}_S \ \mathbf{ref}$ and $\mathbf{t}_2 : \mathbf{T}_S$. We may apply the induction hypothesis to the first subderivation, obtaining $\mathbf{t}'_1 : \mathbf{T}_S \ \mathbf{ref}$. Thus, by T-ASSIGN, $\mathbf{t}'_1 := \mathbf{t}_2 : \mathbf{unit}$ and so $\mathbf{t}' : \mathbf{unit}$.

Subcase E-ASSIGN2:

$$\begin{aligned} \mathbf{t}_2 \mid \mu &\longrightarrow \mathbf{t}'_2 \mid \mu' \\ \mathbf{t}' &= \mathbf{v}_1 := \mathbf{t}'_2 \mid \mu' \end{aligned}$$

From the assumptions of the T-ASSIGN case, we have subderivations of the original typing derivation with

conclusions $\mathbf{v}_1 : \mathbf{T}_S \ \mathbf{ref}$ and $\mathbf{t}_2 : \mathbf{T}_S$. We may apply the induction hypothesis to the second subderivation, obtaining $\mathbf{t}'_2 : \mathbf{T}_S$. Thus, by T-ASSIGN, $\mathbf{v}'_1 := \mathbf{t}'_2 : \mathbf{unit}$ and so $\mathbf{t}' : \mathbf{unit}$.

Subcase E-ASSIGN:

$$\begin{aligned} \mathbf{t}_1 &= l \\ \mathbf{t}_2 &= \mathbf{v}_2 \\ \mathbf{t}' &= \mathbf{unit} \mid [l \mapsto \mathbf{v}_1] \mu \end{aligned}$$

Immediate since $\mathbf{t}' = \mathbf{unit}$.

Case T-DEREF:

$$\begin{aligned} \mathbf{t} &= \$\mathbf{t}_1 \\ \mathbf{T} &= \mathbf{T}_S \end{aligned}$$

If the last rule in the derivation is T-DEREF, then we know from the form of this rule that \mathbf{t} must have the form $\$\mathbf{t}_1$ and that \mathbf{T} is \mathbf{T}_S . We must further have a subderivation with conclusion $\mathbf{t}_1 : \mathbf{T}_S \ \mathbf{ref}$ for some \mathbf{T}_S . Looking at the evaluation rules with the dereferencing operator on the left-hand side, there are two rules by which $\mathbf{t} \longrightarrow \mathbf{t}'$ can be derived: E-DEREF and E-DEREFLOC.

Subcase E-DEREF:

$$\begin{aligned} \mathbf{t}_1 \mid \mu &\longrightarrow \mathbf{t}'_1 \mid \mu' \\ \mathbf{t}' &= \$\mathbf{t}'_1 \mid \mu' \end{aligned}$$

From the assumptions of the T-DEREF case, we have a subderivation of the original typing derivation with conclusion $\mathbf{t}_1 : \mathbf{T}_S \ \mathbf{ref}$. We may apply the induction hypothesis, obtaining $\mathbf{t}'_1 : \mathbf{T}_S \ \mathbf{ref}$. Thus, by T-DEREF, $\$\mathbf{t}'_1 : \mathbf{T}_S$ and so $\mathbf{t}' : \mathbf{T}_S$.

Subcase E-DEREFLOC:

$$\begin{aligned} \mathbf{t}_1 &= l \mid \mu \\ \mathbf{t}' &= \$l \mid \mu \end{aligned}$$

From the assumptions of the T-DEREF case, we have a subderivation of the original typing derivation with conclusion $\mathbf{t}_1 : \mathbf{T}_S \ \mathbf{ref}$. In this subcase we have that \mathbf{t}_1 is the location value l . We know that l has type $\mathbf{T}_S \ \mathbf{ref}$. Thus, by T-ASSIGN, $\$l : \mathbf{T}_S$ and so $\mathbf{t}' : \mathbf{T}_S$.

Case T-SW-H:

$$\begin{aligned} \mathbf{t} &= \mathbf{sw} \ \mathbf{t}_1 \\ \mathbf{T} &= \mathbf{T}_H \ \mathbf{sw} \end{aligned}$$

If the last rule in the derivation is T-SW-H, then we know from the form of this rule that \mathbf{t} must have the form $\mathbf{sw} \ \mathbf{t}_1$ and that \mathbf{T} is $\mathbf{T}_H \ \mathbf{sw}$ for some \mathbf{T}_H . We must further have a subderivation with conclusion $\mathbf{t}_1 : \mathbf{T}_H$. Looking at the evaluation rules with \mathbf{sw} on the left-hand side, there is one rule by which $\mathbf{t} \longrightarrow \mathbf{t}'$ can be derived: E-SW. By the induction hypothesis, $\mathbf{t}'_1 : \mathbf{T}_H$ and so by T-SW-H, $\mathbf{sw} \ \mathbf{t}'_1 : \mathbf{T}_H \ \mathbf{sw}$. Therefore, $\mathbf{t}' : \mathbf{T}_H \ \mathbf{sw}$ as needed.

Case T-SW-M:

Similar to T-SW-H.

Case T-HW-H:

$$\begin{aligned} \mathbf{t} &= \mathbf{hw} \ \mathbf{t}_1 \\ \mathbf{T} &= \mathbf{T}_H \end{aligned}$$

If the last rule in the derivation is T-HW-H, then we know from the form of this rule that \mathbf{t} must have the form $\mathbf{hw} \ \mathbf{t}_1$ and that \mathbf{T} is \mathbf{T}_H for some \mathbf{T}_H . We must further have a subderivation with conclusion $\mathbf{t}_1 : \mathbf{T}_H \ \mathbf{sw}$. Looking at the evaluation rules with \mathbf{hw} on the left-hand side, there is one rule by which $\mathbf{t} \longrightarrow \mathbf{t}'$ can be derived: E-HW. By the induction hypothesis, $\mathbf{t}'_1 : \mathbf{T}_H \ \mathbf{sw}$ and so by T-SW-H, $\mathbf{hw} \ \mathbf{t}'_1 : \mathbf{T}_H$. Therefore, $\mathbf{t}' : \mathbf{T}_H$ as needed.

Case T-HW-M:

Similar to T-HW-H.

Case T-ARR-ACC:

$$\begin{aligned} \mathbf{t} &= \mathbf{t}_1[\mathbf{t}_2] \\ \mathbf{T} &= \mathbf{T}_H \end{aligned}$$

If the last rule in the derivation is T-ARR-ACC, then we know from the form of this rule that \mathbf{t} must have the form $\mathbf{t}_1[\mathbf{t}_2]$ and that \mathbf{T} is \mathbf{T}_H . We must further have a subderivation with conclusion $\mathbf{t}_1 : \mathbf{T}_H[\mathbf{n}]$ and $\mathbf{t}_2 : \mathbf{int}$. Looking at the evaluation rules with array access on the left-hand side, there are two rules by which $\mathbf{t} \longrightarrow \mathbf{t}'$ can be derived: E-ARR-ACC and E-ARR-ACC1.

Subcase E-ARR-ACC:

$$\begin{aligned} \mathbf{t}_1 &= \mathbf{v}_1 \\ \mathbf{t}_2 &= \mathbf{v}_2 \\ \mathbf{t}' &= \mathbf{v}_{1, \mathbf{v}_2} \end{aligned}$$

From the assumptions of the T-ARR-ACC case, we have subderivations of the original typing derivation with conclusions $\mathbf{t}_1 : \mathbf{T}_H[\mathbf{n}]$ and $\mathbf{t}_2 : \mathbf{int}$. Thus, the \mathbf{v}_2^{th} element of \mathbf{v}_1 is of type \mathbf{T}_H , and so $\mathbf{t}' : \mathbf{T}_H$.

Subcase E-ARR-ACC1:

$$\begin{aligned} \mathbf{t}_1 &\longrightarrow \mathbf{t}'_1 \\ \mathbf{t}' &= \mathbf{v}_1[\mathbf{t}'_2] \end{aligned}$$

From the assumptions of the T-ARR-ACC case, we have subderivations of the original typing derivation with conclusions $\mathbf{t}_1 : \mathbf{T}_H[\mathbf{n}]$ and $\mathbf{t}_2 : \mathbf{int}$. By the induction hypothesis, $\mathbf{t}'_2 : \mathbf{int}$. Therefore, by T-ARR-ACC, $\mathbf{v}_1[\mathbf{t}'_2] : \mathbf{T}_H$ and so $\mathbf{t}' : \mathbf{T}_H$.

Case T-RCD:

$$\begin{aligned} \mathbf{t} &= \{\mathbf{l}_i = \mathbf{t}_i^{i \in 1..n}\} \\ \mathbf{T} &= \{\mathbf{l}_i : \mathbf{T}_i^{i \in 1..n}\} \end{aligned}$$

If the last rule in the derivation is T-RCD, then we know from the form of this rule that \mathbf{t} must have the form $\{\mathbf{l}_i = \mathbf{t}_i^{i \in 1..n}\}$ and that \mathbf{T} is $\{\mathbf{l}_i : \mathbf{T}_i^{i \in 1..n}\}$. We must further have a subderivation for each i with conclusion $\mathbf{t}_i : \mathbf{T}_i$. Looking at the evaluation rules with record creation on the left-hand side, there are two rules by which $\mathbf{t} \longrightarrow \mathbf{t}'$ can be derived: E-RCD and E-RCDV.

Subcase E-RCD:

$$\begin{aligned} \mathbf{t}_j &\longrightarrow \mathbf{t}'_j \\ \mathbf{t}' &= \{\mathbf{l}_i = \mathbf{v}_i^{i \in 1..j-1}, \mathbf{l}_j = \mathbf{t}'_j, \mathbf{l}_k = \mathbf{t}_k^{k \in j+1..n}\} \end{aligned}$$

From the assumptions of the T-RCD case, we have for each i a subderivation of the original typing derivation with conclusion $\mathbf{t}_i : \mathbf{T}_i$. Thus, $\mathbf{t}_j : \mathbf{T}_j$. We may apply the induction hypothesis, obtaining $\mathbf{t}'_j : \mathbf{T}_j$. Thus, by T-RCD, we obtain $\{\mathbf{l}_i = \mathbf{v}_i^{i \in 1..j-1}, \mathbf{l}_j = \mathbf{t}'_j, \mathbf{l}_k = \mathbf{t}_k^{k \in j+1..n} : \{\mathbf{l}_i : \mathbf{T}_i^{i \in 1..n}\}$ and so $\mathbf{t}' : \{\mathbf{l}_i : \mathbf{T}_i^{i \in 1..n}\}$.

Subcase E-RCDV:

$$\begin{aligned} \mathbf{t}_j &= \mathbf{v}_j \\ \mathbf{t}' &= \{\mathbf{l}_i = \mathbf{v}_i^{i \in 1..j}, \mathbf{l}_k = \mathbf{t}_k^{k \in j+1..n}\} \end{aligned}$$

If $\mathbf{t} \longrightarrow \mathbf{t}'$ is derived using E-RCDV, then from the form of this rule we see that $\mathbf{t}_j = \mathbf{v}_j$ and the resulting term \mathbf{t}' is an evaluation on the remaining fields of the record. This means we are finished, since we know (by the assumptions of the T-RCD case) that $\mathbf{t}' : \{\mathbf{l}_i = \mathbf{v}_i^{i \in 1..j-1}, \mathbf{l}_j = \mathbf{t}'_j, \mathbf{l}_k = \mathbf{t}_k^{k \in j+1..n}\}$, which is what we need.

Case T-PROJ:

$$\begin{aligned} \mathbf{t} &= \# \mathbf{l}_j \ \mathbf{t}_1 \\ \mathbf{T} &= \mathbf{T}_j \end{aligned}$$

If the last rule in the derivation is T-PROJ, then we know from the form of this rule that t must have the form $\#l_j \ t_1$ and that T is T_j . We must further have a subderivation with conclusion $t_1 : \{l_i : T_i^{i \in 1..n}\}$. Looking at the evaluation rules with projection on the left-hand side, there are two rules by which $t \rightarrow t'$ can be derived: E-PROJ-RCD and E-PROJ.

Subcase E-PROJ-RCD:

$$\begin{aligned} t_1 &= \#l_j \ \{l_i = v_i^{i \in 1..n}\} \\ t' &= v_j \end{aligned}$$

From the assumptions of the T-PROJ case, we have a subderivation of the original typing derivation with conclusion $t_1 : \{l_i : T_i^{i \in 1..n}\}$. Thus extracting the value corresponding to field l_j must yield v_j which is of type T_j , and so $t' : T_j$.

Subcase E-PROJ:

$$\begin{aligned} t_1 &\rightarrow t'_1 \\ t' &= \#l_j \ t'_1 \end{aligned}$$

From the assumptions of the T-PROJ case, we have a subderivation of the original typing derivation with conclusion $t_1 : \{l_i : T_i^{i \in 1..n}\}$. We may apply the induction hypothesis, obtaining $t'_1 : \{l_i : T_i^{i \in 1..n}\}$. Thus, by T-PROJ, $\#l_j \ t'_1 : T_j$ and so $t' : T_j$.

Case T-LET:

$$\begin{aligned} t &= \text{let } (x_i = t_i)^{i \in 1..n} \text{ in } t_0 \text{ end} \\ T &= T_0 \end{aligned}$$

If the last rule in the derivation is T-LET, then we know from the form of this rule that t must have the form $\text{let } (x_i = t_i)^{i \in 1..n} \text{ in } t_0 \text{ end}$ and that T is T_0 . We must further have subderivation with conclusions $t_1 : T_1$ and $\text{let } (x_i = t_i)^{i \in 2..n} \text{ in } t_0 \text{ end} : T_0$. Looking at the evaluation rules with let-bindings on the left-hand side, there are three rules by which $t \rightarrow t'$ can be derived: E-LET, E-LETV1, and E-LETV2.

Subcase E-LET:

$$\begin{aligned} t_1 &\rightarrow t'_1 \\ t' &= \text{let } x_1 = t'_1 \ (x_i = \\ &\quad t_i)^{i \in 1..n} \text{ in } t_0 \text{ end} \end{aligned}$$

From the assumptions of the T-LET case, we have a subderivation of the original typing derivation with conclusion $t_1 : T_1$. We may apply the induction hypothesis, obtaining $t'_1 : T_1$. Thus, by T-LET, $\text{let } x_1 = t'_1 \ (x_i = t_i)^{i \in 1..n} \text{ in } t_0 \text{ end} : T_0$ and so $t' : T_0$.

Subcase E-LETV1:

$$\begin{aligned} t_1 &= v_1 \\ t' &= \text{let } x_2 = t_2 \ (x_i = \\ &\quad t_i)^{i \in 3..n} \text{ in } t_0 \text{ end} \end{aligned}$$

If t_1 is v_1 we simply augment the variable and type binding stores with the mapping $x_1 \mapsto v_1$. The term being evaluated in the body of the let-binding does not alter and as such $t' : T_0$.

Subcase E-LETV2:

Similar to E-LETV1.

Case T-DATATY:

$$\begin{aligned} t &= C_j \ t_0 \\ T &= \langle C_i : T_i \rangle^{i \in 1..n} \end{aligned}$$

If the last rule in the derivation is T-DATATY, then we know from the form of this rule that t must have the form $C_j \ t_0$ and that T is $\langle C_i : T_i \rangle^{i \in 1..n}$. We must further have a subderivation with conclusion $t_0 : T_j$. Looking at the evaluation rules with datatype construction on the left-hand side, there is one rule by which $t_0 \rightarrow t'_0$ which is E-DATATY. By the induction hypothesis, $t'_0 : T_j$. Then, by T-DATATY, it follows that $C_j \ t'_0 : \langle C_i : T_i \rangle^{i \in 1..n}$. Therefore, $t' : \langle C_i : T_i \rangle^{i \in 1..n}$.

Case T-CASE:

$$\begin{aligned} t &= \text{case } t_0 \text{ of } C_i \ x_i \Rightarrow t_i^{i \in 1..n} \\ T &= T' \end{aligned}$$

If the last rule in the derivation is T-CASE, then we know from the form of this rule that t must have the form $\text{case } t_0 \text{ of } C_i \ x_i \Rightarrow t_i^{i \in 1..n}$ and that T is T' for some T' . We must further have subderivations with conclusions $t_0 : \langle C_i : T_i \rangle^{i \in 1..n}$ and, for each i , $t_i : T'$. Looking at the evaluation rules with case expressions on the left-hand side, there are two rules by which $t_0 \rightarrow t'_0$ can be derived: E-CASE and E-CASE-TY.

Subcase E-CASE:

$$\begin{aligned} t_0 &\rightarrow t'_0 \\ t' &= \text{case } t'_0 \text{ of } C_i \ x_i \Rightarrow \\ &\quad t_i^{i \in 1..n} \end{aligned}$$

From the assumptions of the T-CASE case, we have a subderivation of the original typing derivation with conclusion $t_0 : \langle C_i : T_i \rangle^{i \in 1..n}$. We may apply the induction hypothesis, obtaining $t'_0 : \langle C_i : T_i \rangle^{i \in 1..n}$. Thus, by T-

CASE, case t'_0 of $C_i \ x_i \Rightarrow t_i^{i \in 1..n}$
 $: T_0$, and so $t' : T'$. □

Subcase E-CASE-TY:

$$\begin{aligned} t_0 &= C_j \ v_j \\ t' &= [x_j \mapsto v_j] \ t_j \end{aligned}$$

As we assumed earlier, $t_i : T'$ for all t_i
and so $t_j : T'$. As a result, $[x_j \mapsto v_j]$
 $t_j : T'$.

Theorem 5 (Safety). *A well-typed term can never reach a stuck state during evaluation.*

Proof. The progress theorem demonstrates that a well-typed term is not stuck, and the preservation theorem shows that if a well-typed term takes a step of evaluation, then the resulting term is also well-typed. In combination, these guarantee safety. □