**GEMINI**

*Designing and Implementing a Functional Hardware Description Language*

*Author Advisor*

Aditya Srinivasan Dr. Andrew D. Hilton



Department of Electrical & Computer Engineering

DUKE UNIVERSITY

A thesis submitted to DUKE UNIVERSITY in accordance with the requirements of Graduation with Departmental Distinction in ELECTRICAL & COMPUTER ENGINEERING in the Pratt School of Engineering.

APRIL 2018

**ABSTRACT**

Hardware description languages have revolutionized the ability to scalably design, verify, simulate, and synthesize electronic circuits. While HDLs have provided more powerful features in each successive iteration, they still lack the expressivity and modularity characteristic of higher-level programming languages. In order to address this, I undertook a significant software development project to design an HDL named Gemini, influenced by functional programming languages, and to implement a compiler that accepts a Gemini program and produces the appropriate Verilog code. Gemini is unlike any other programming language for three primary reasons: the existence of multiple kinds, type parameterization with values, and the manifestation of time as a language feature. The implementation is grounded in results from type theory to guarantee safety of the type system and correctness of evaluation.

**ACKNOWLEDGEMENTS**

I would like to express my sincere gratitude to my advisor Dr. Hilton for his involvement and support throughout the course of this project. His immense knowledge and passion for type theory and compilers encouraged me to pursue an ambitious endeavor and approach it with great excitement and determination.

**TABLE OF CONTENTS**

**1 Introduction 3**

**2 Language Specification 3**

**2.1** Values **3**

**2.2** Expressions **4**

**2.3** Types **5**

**2.4** Declarations **6**

**2.5** Library **7**

**3 Formalizations ?**

**3.1** Software Grammar **?**

**3.2** Hardware Grammar **?**

**3.3** Derived Terms **?**

**3.4** Typing Rules **?**

**3.5** Evaluation Rules **?**

**4 Type Safety ?**

**4.1** Proof of Progress **?**

**4.2** Proof of Preservation **?**

**4.3** Proof of Safety **?**

**5 Implementation ?**

**5.1** Lexer **?**

**5.2** Parser **?**

**5.3** Semantic Analysis **?**

**5.3.1** Type Decoration **?**

**5.3.2** Type Inference **?**

**5.3.3** Type-Checking **?**

**5.4**  Evaluation **?**

**5.5** Verilog Production **?**

**6 Conclusion ?**

**7 References ?**

**A Appendix A ?**

**B Appendix B ?**

**C Appendix C ?**

**D Appendix D ?**

**1** **INTRODUCTION**

Hardware description languages (HDLs) have existed since the 1960s as powerful tools to precisely represent the design of electrical components [1]. As very-large-scale-integration (VLSI) became more popular for design, the first modern HDL, Verilog, was introduced between 1983 and 1984 [2]. At around the same time, the Department of Defense began developing a new standard named VHDL [3].

Since their inception, these two HDLs and variant descendent languages have been among the most widely-used in industry for the verification, synthesis, and simulation of circuits. While successive iterations have introduced more powerful features, such as datatypes and strong type systems, they have not been able to keep up with the ways in which software programming languages have evolved. This hinders their ability to concisely and efficiently express the designs they are intended to represent.

The design of Gemini was motivated by the need to provide higher-level abstractions to hardware description languages in order to improve expressivity and modularity. Such features include a strong type system, parametric polymorphism, variant types, higher-order functions, recursive types, and recursion, which are borrowed from popular contemporary functional programming languages.

**2 LANGUAGE SPECIFICATION**

In this section, we will informally specify various features of Gemini with examples of syntax. Section 3 will provide formalizations of these.

**2.1 VALUES**

All programming languages compute on values. A value is an atomic unit of data that cannot be reduced or evaluated any further. Common examples of values in programming languages are integer literals (123), boolean literals (true), and string literals (“abc”).

In Gemini, there are four types of *hardware values*: bits, arrays, records, and tuples.All hardware values are arrangements of bits. Elements in an array or record may be of any hardware type, thus supporting multidimensional arrays. Record values have explicitly labeled fields, whereas tuples have implicitly numbered fields corresponding to the index.

In Gemini, there are several types of *software values*: integers, reals, strings, lists, records, tuples, references, software-wrappers, variants, and functions. We devote a special discussion to the software-wrapper value in Section 2.3.

In Gemini, there is one type of *module value*, which is similar to a function except it operates on hardware values. Further, higher-order modules are unsupported. This is intentional by design, as will be discussed in Section 2.3.

The syntax for declaring each value is formalized in Section 3.

**2.2 EXPRESSIONS**

Expressions operate on values and subexpressions, and can be reduced by evaluation rules to other expressions or values. These evaluation rules will be formalized in Section 3.4; here we give an overview.

**2.2.1 Operators**

Operators are analogous to functions, except their application is infix and the arguments are the operands. In Gemini, operator overloading is not as common as in other languages. For example, the addition of integers and reals involve separate operators (+ and +. respectively). This is common in strongly typed languages, since it enables the exact determination of types when performing inference. For a complete list of the operators and their semantic results, please consult Figures A-1 through A-5 of Appendix A.

**2.2.2 Accesses**

Records (both software and hardware), tuples (both software and hardware), arrays, and refs can be accessed to retrieve data. Record accesses are performed with the syntax #f e where f is the field name and e is the record to access. Tuple accesses are performed similarly, except f must be an integer literal corresponding to the field index. Array accesses are made with the syntax e[:i:], where e is the array and i is the index to access. Reference accesses are made with the syntax $e, where e is the expression to dereference.

**2.2.3 Conditionals**

Conditional expressions enable control flow within a program. There are two types of conditionals: if-then-else expressions and if-then expressions. The former is produced with the syntax **if** e1 **then** e2 **else** e3, and the result is e2 if e1 is nonzero, else it is e3. The latter is a special case where the else clause is omitted and implicitly returns the empty tuple: **if** e1 **then** e2.

**2.2.4 Assignment**

Assignments provide values to references. The syntax is e1 := e2, where e1 is the reference.

**2.2.5 Sequence**

Sequence expressions allow multiple expressions to be evaluated for side-effect, with the last one being returned as the value. They are made with syntax (e1; e2; en-1; en).

**2.2.6 Pattern-match**

Pattern-match expressions are used as another, more general form of control flow. A pattern-match expression consists of a test expression e and an ordered set of match-result pairs (m1, r1), …, (mn, rn). The value of e is compared to each match, and the first mi for which there is a match, ri is returned. The syntax is **case** e **of** m1 => r1 |: m2 => r2 |: mn-1 => rn-1 |: mn => rn.

**2.2.6 Let-bindings**

Let-bindings are used to bind identifiers with values or types and to then evaluate an expression in the same scope. The syntax is **let** **in** e **end**, where is a series of zero or more declarations, which will be discussed further in Section 2.4.

**2.3 TYPES**

In the previous discussion we have mentioned the concept of types without providing a formal definition. A type is a category to which a value can be assigned. Alternatively, a type is a set of values, with each type being disjoint from the others.

In type theory, there is a special term given to the type of types: kinds. Kinding is necessary since not all types have the same structure. For example, the type list is itself not a type, but a type constructor. It is a type-level function that takes a type, such as int, and produces a type, such as int list.

In conventional type systems, kinds are built from an atomic kind, written as and pronounced “type”, and the constructor [5]. The kind of proper types – such as int, real, and string – is , the kind of type constructors – such as list and ref – is .

In Gemini, the kinding system is instead composed of three atomic kinds: (the software kind), (the hardware kind), and (the module kind). The reason for the separation is to enforce the trichotomy between software, hardware, and module types in the type system, since types from two different kinds are not interchangeable. The BNF diagram in Figure 2-1 lists the member types of each kind.

**Figure 2-1**: Kind definitions

::= int

| real

| string

| list

| {l1: , ..., ln: }

| ref

| sw

| Ci

|

::= bit

| []

| @

| #{l1: , ..., ln: }

::=

Here, and refer to types belonging to kind and respectively. The kinding definition of Gemini intentionally prohibits certain expressions and values from being defined. For example, higher-order modules cannot exist since the module type only allows types of kind as the argument and result. Further, hardware values cannot persist in software values such as lists and functions, and vice versa. This is an important property that enables the compilation of Gemini by first evaluating all software expressions and then processing the resulting program consisting only of hardware values.

We note that records and tuples are both represented by the same type. This is because tuples are a special kind of record with implicitly numbered fields, and apart from syntax these two are treated as the same.

In addition to having three atomic kinds, the Gemini type system is interesting for three more reasons. Firstly, the concept of time is modeled by the type system, something which few other software programming languages do. This renders values of different temporal types incompatible, which is necessary for generating correct hardware. Secondly, array types and temporal types are parameterized by some integer . These differ from type constructors such as lists, which accept a type to produce a new type, in that they accept a value to produce a new type. This is necessary since hardware arrays of differing lengths represent unique types, as do temporal values of differing periods. Thirdly, the software-wrapper is a vital operator allowing for inter-kind manipulation of data. When a hardware value is wrapped, it can be persisted in any software data type, such as lists, records, or functions. The actual value can never be read, and the operator unsw unwraps the value for it to be used as a hardware value.

**2.4 DECLARATIONS**

There are five types of declarations that can be made: values, functions, types, datatypes, and modules. Declarations can be made in structures or the **let**...**in** section of let-bindings.

**2.4.1** **Values**

Value declarations bind an identifier x to a value with one of two syntaxes. The value can be declared implicitly using the syntax **val** x = e. Alternatively, the value can be declared with an explicit type using the syntax **val** x : T = e, for some type T.

**2.4.2 Functions**

Function declarations bind an identifier f to a function with the syntax **fun** f arg1 argn = e, where argi can be an identifier, a record of arguments, or a tuple of arguments. The function and any arguments may be explicitly typed by suffixing with ‘: T’ for some type T.

**2.4.3 Types**

Type declarations bind an identifier t to a type with the syntax **type** t = T, for some type T. Type constructors that are parametrically polymorphic may also be declared by introducing type variables using the syntax **type** ‘a t = T’a, where T’a is some type that may reference ’a. The syntax for type declarations is shown in Figure A-6 of Appendix A.

**2.4.4 Datatypes**

Datatype declarations bind an identifier d to a variant type. Software datatypes are declared with the syntax **sdatatype** d = C1 **of** TS1 |: Cn **of** TSn, where Ci is a datatype constructor that accepts software type TSi. Similarly, hardware datatypes are declared by **hdatatype** d = C1 **of** TH1 |: Cn **of** THn. Constructors need not require any type to be constructed, in which case the ‘**of** Ti’ is omitted.

**2.4.5 Modules**

Module declarations bind an identifier m to a module with the syntax **module** m arg = e, where arg can be an identifier, a hardware record of arguments, or a hardware tuple of arguments. The module and argument may be explicitly typed by suffixing with ‘: T’ for some type T. Modules can also be parameterized with software values using angle-bracket notation: **module** m <param> arg = e.

**2.5 LIBRARY**

The Gemini library provides useful built-in functions and modules, for operating on software and hardware values respectively. A complete list of these can be found in Figure A-7 of Appendix A.

**3 FORMALIZATIONS**

Having specified the type system and syntax, we now formalize the language grammar and rules.

**3.1 SOFTWARE GRAMMAR**

We first define the grammars for software-typed values and expressions.

**3.1.1 Software Value Grammar**

An excerpt of the grammar is shown below. Figure B-2 of Appendix B illustrates the grammar in full.

**Figure 3-1**: Excerpt of software value grammar

::=

|

|

|

|

|

|

|

|

::=

::=

This grammar represents each value abstractly in a mathematical sense instead of capturing the syntax.

**3.1.2** **Software Expression Grammar**

Here we specify the grammar for the syntax of software values and expressions. An excerpt of the grammar is shown below. Figure B-3 of Appendix B illustrates the grammar in full.

**Figure 3-2**: Excerpt of software expression grammar

::=

|

|

|

|

|

|

|

|

::=

|

|

|

|

|

|

|

::= ref

**3.1.3 Derived Expressions**

Some expressions can be expressed in terms of others, and are therefore *derived*. Beyond the syntactic layer of the compiler, these are treated analogously to the expressions from which they are derived, thereby simplifying the implementation of later phases. The set of derived terms can be found in Figure B-1 of Appendix B.

**3.2 HARDWARE GRAMMAR**

There exists both software-typed values and expressions. However, there only exists hardware-typed values. The meaning of a hardware-typed expression is ill-defined since no hardware circuit can be reduced any further than the structure it takes. Here we define the grammars for hardware values.

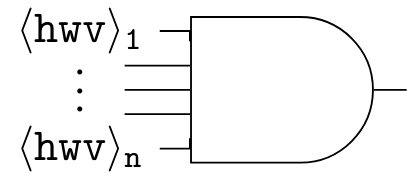
**3.2.1 Hardware Value Grammar**

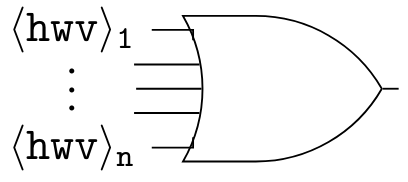
The grammar in full is shown below.

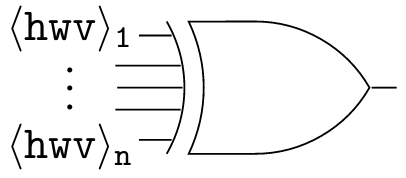
**Figure 3-3**: Hardware value grammar

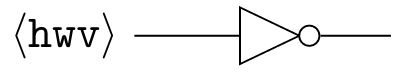
::=

|









|

| dff()

**3.2.2** **Hardware Syntax Grammar**

An excerpt of the syntax grammar is shown below. Figure B-4 of Appendix B shows the grammar in full.

**Figure 3-4**: Excerpt of hardware syntax grammar

::=

|

|

|

|

::=

|

|

::= ‘b:

The grammar here is meant to augment the grammar shown in Figure 3-2. Certain non-terminals are repeated, and any new rules appearing here are appended to those from earlier. The grammar itself is not responsible for enforcing typing rules, which brings us to the next section.

**3.4 TYPING RULES**

With the grammar formalized, it is possible to identify whether a program is gramatically valid and to construct an abstract syntax tree by parsing it. This section provides the tools to verify that the program is well-typed, which is imperative for correctness in compilation.

Each typing rule is a theorem, with a set of propositions or hypotheses and a conclusion. Diagramatically, the antecedent clause is written above the horizontal line, with the consequent clause below.

An excerpt of the set of typing rules is shown below. Figure B-5 of Appendix B illustrates the set of typing rules in completion.

**Figure 3-5**: Excerpt of typing rules

|  |  |
| --- | --- |
| x : T  (T-VAR)  x : T | , x : T1 t2 : T2  (T-ABS)  x:T1.t2 : T1 T2 |
| t1:T1 T2 t2:T1  (T-APP)  t1t2 : T2 | t1:int t2:int  (T-INT-ADD)  t1 + t2 : int |
| t1 : TH t2 : TH  (T-AND)  t1 & t2 : TH | t1:int t2:T t3:T  (T-IF)  if t1 then t2 else t3 : T |

As an example, rule T-IF is pronounced “if t1 has type int, t2 has type T, and t3 has type T, then if t1 then t2 else t3 has type T”. Note in this example that the then- and else-clause may have type T belonging to any kind, whereas typing rule T-AND restricts t1 and t1 to a type of kind hardware.

**3.5 EVALUATION RULES**

Now having a rigorous formulation of the syntax and typing rules of our language, we need to precisely define how expressions are evaluated. This is known as defining the semantics of the language, for which there are three basic approaches: *operational semantics*, *denotational semantics*, and *axiomatic semantics* [5]. In this paper, we use operational semantics for its simplicity and flexibility.

Operational semantics define an abstract state machine, where each state is an expression. The machine’s behavior is defined by a transition function that either yields the next state by performing a step of computation, or declares that the machine has halted by reaching some terminal value. Operational semantics can be further partitioned into small-step and big-step semantics. Small-step semantics, or structural operational semantics, consider how evaluation takes place one step at a time. Big-step semantics, or natural semantics, instead describe the final value to which some expression evaluates [5].

In general, big-step semantics are less verbose since intermediate expression states need not be encoded in the machine behavior. However, small-step semantics are more precise and readily translatable for implementation, and since our goal is developing a compiler, we choose to use small-step semantics.

Similarly to typing rules, each evaluation rule is a theorem. An excerpt of the set of evaluation rules is shown below. Figure B-6 of Appendix B illustrates the set of evaluation rules in completion.

**Figure 3-6**: Excerpt of evaluation rules

|  |  |
| --- | --- |
| t1 t1’  (E-APP1)  t1 t2 t1’ t2 | v1 0 (E-IFELSE-T)    if v1 then t2 else t3 t2 |
| t2 t2’  (E-APP2)  v1 t2 v1 t2’ | (E-IFELSE-F)  if 0 then t2 else t3 t3 |
| (x.t1)v1 [xv1]t1 (E-APPABS) | t1 t1’ (E-IFELSE)    if t1 then t2 else t3  if t1’ then t2 else t3 |

As an example, rule E-IFELSE is pronounced “if t1 evaluates to t1’ in one step, then the whole expression if t1 then t2 else t3 evaluates to if t1’ then t2 else t3”.

The evaluation rules together define a precise evaluation strategy. For example, consider the expression if x then (if 1 then “a” else “b”) else “c”. Under the evaluation rules of our language, it is not possible for this to evaluate to if x then “a” else “c”, despite this being a state that would evaluate to an equivalent value. We must first evaluate the guard of the outer-conditional by rule E-IFELSE. Once it is a value, then we pick one of the then- and else-clause based on rules E-IFELSE-T and E-IFELSE-F and compute on it. A useful property is the determinacy of one-step evaluation, stating that if t t’ and t t’’, then t’ = t’’. This ensures that evaluation is a deterministic process.

**4 TYPE SAFETY**

Given formalizations of the grammar, semantics, and evaluation strategy of the language, it is desirable to prove a basic property of Gemini’s type system: *safety*. First, we must define a few terms.

**DEFINITION**: A term t is in *normal form* if no evaluation rule applies to it. (D-1)

**DEFINITION**: A term t is in a *stuck state* if it is in normal form but it is not a value. (D-2)

**DEFINITION**: A type system possesses *safety* if a well-typed term can never reach a stuck state during evaluation. (D-3)

We demonstrate the safety of our type system with two propreties.

**DEFINITION**: A type system has the property of *progress* if a well-typed term is never in a stuck state; either it is a value or it can take a step according to some evaluation rule. (D-4)

**DEFINITION**: A type system has the property of *preservation* if when a well-typed term takes an evaluation step, the resulting term is also well-typed. (D-5)

In order to prove the progress theorem, we first posit two lemmas.

**LEMMA**: The following are true, and constitute the inversion of the typing relation: (L-1)

1. If x : R, then x : R
2. If x:T1.t2:R, then R = T1 R2 for some R2 with ,x:T1 t2:R2
3. If t1 t2:R then there is some type T11 such that t1:T11 R and that t2:T11
4. If : R, then R = int

The remainder of the cases are omitted here and are shown in full in Figure C-1 of Appendix C.

*Proof*: Immediate from the definition of the typing rules.

**LEMMA**: The following are true, and constitute the canonical forms: (L-2)

1. If v is a value of type int, then v is an integer value according to the software value grammar.
2. If v is a value of type real, then v is a real value according to the software value grammar.
3. If v is a value of type string, then v is a string value according to the software value grammar.
4. If v is a value of type bit, then v is either 0 or 1.
5. If v is a value of type T1 T2, then v = T1.t2.
6. If v is a value of type TS ref, then v is a location in store .
7. If v is a value of type {li : Tii 1..n}, then v is a value with the form {li = vii 1..n}.
8. If C is a constructor of datatype D accepting type T1 and v is a value of type T1 then C v is a value of type D with form .
9. If v is a value of type TH[n], then v is a value according to the hardware value grammar.
10. If v is a value of type #{li : Tii 1..n}, then v is a value with the form #{li = vii 1..n}.
11. If v is a value of type TH sw, then v is a value with the for some vH of type TH.

*Proof*: We refer to the first 10 cases of the Lemma L-1 as they pertain to values in this language.

*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 cases in consideration from Lemma L-1, only case 4 tells us that the value has type int.

The remaining cases are similar.

**4.1 PROOF OF PROGRESS**

We are now equipped to prove the theorem of progress

**THEOREM OF PROGRESS**: Suppose t is a closed, well-typed term ( t : T for some T). Then either t is a value or else there is some t’ with t t’. (TH-1)

*Proof*: By structural 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-APP:

t = t1 t2

t1 : T11 T12

t2 : T12

By the induction hypothesis, either t1 is a value or else there is some other t1’ for which t1  t1’, and likewise for t2. If t1 t1’ then by E-APP1, t t1’ t2. On the other hand, if t1 is a value and t2 t2’, then by E-APP2, t t1 t2’. Finally, if both t1 and t2 are values, then case 5 of the canonical forms lemma tells us that t1 has the form : T11.t12 and so by E-APPABS, t [xt2]t12 which is a value.

The remaining cases are shown in full in Proof C-2 of Appendix C.

**4.2 PROOF OF PRESERVATION**

**THEOREM OF PRESERVATION**: If t : T and t t’, then t’ : T. (TH-2)

*Proof*: By structural 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 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

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 t’ for any t’, and the requirements of the theorem are vacuously satisfied.

*Case* T-APP:

t = t1 t2

t1 : T11 T12

t2 : T11

T = T12

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

*Subcase* E-APP1:

t1 t1’

t’ = t1’ t2

From the assumptions of the T-APP case, we have a subderivation of the original typing derivation whose conclusion is t1 : T11 T12. We can apply the induction hypothesis to this subderivation obtaining t1’ : T11 T12. Combining this with the fact that t2 : T11, we can apply rule T-APP to conclude that t’: T.

*Subcase* E-APP2:

Similar to E-APP1.

*Subcase* E-APPABS:

t1 = : T11.t12

t2 = v2

t’ = [xv2]t12

Using Lemma L-1, we can desconstruct the typing derivation for : T11.t12 yielding : T11 t12 : T12. From this we obtain t’ : T12.

The remaining cases are shown in full in Proof C-3 of Appendix C.

**4.3 PROOF OF SAFETY**

**THEOREM OF SAFETY**: A well-typed term can never reach a stuck state in evaluation. (TH-3)

*Proof*: Theorem TH-1 demonstrates that a well-typed term is not stuck, and Theorem TH-2 demonstrates that if a well-typed term takes a step of evaluation, then the resulting term is also well-typed. In combination and inductively, these guarantee safety.

**5 IMPLEMENTATION**

Having formalized the language and proved desirable properties of the type system, we are positioned to implement a compiler. The compiler we discuss in this paper accepts a Gemini program as input and produces a Verilog file as output, and is written in SML-NJ.

**5.1 LEXER**

The first phase of compilation is lexical analysis, performed by the lexer. In this phase, the program is scanned to produce syntactic units called *lexemes*, which are then classified into a particular token class. The lexer is specified by an ordered set of patterns, which match against certain sequences of characters. The lexer scans linearly until it encounters a sequence of characters that matches some pattern. We denote a sequence of characters from position to as . The lexer follows two priority rules:

1. **Rule of longest match**: if the lexer has encountered a valid sequence , it will first check whether is also valid; if it is, then it will disregard in favor of else it tokenizes .
2. **Rule of earliest pattern**: if two lexer rules match the same sequence , then it will tokenize based on the pattern that appears earliest in the list.

For example, if the tokens **>**, **=**, and **>=** all exist in a language, the lexer will tokenize the character sequence ‘>=’ by rule 1, instead of the the token **>** followed by the token **=**. Further, if the keyword **if** exists in a language, and variable identifiers are defined as any alphabetic sequence of characters, then the tokenization of the character sequence ‘if’ will depend on the relative ordering of the patterns. We desire the pattern for keyword **if** to appear before the pattern for identifiers in order to tokenize the character sequence as a keyword by rule 2. If the order were reversed, the lexer would never tokenize the keyword.

In some cases, the lexer needs to perform some basic computation to attach values to tokens. For example, Gemini allows integers to be declared in various bases. When the lexer encounters a hexadecimal representation of an integer, such as #’h:beef, it computes the integer value in base-10 and tokenizes it as an integer value, the same that it would if it encountered the equivalent literal 48879. This allows various representations to be treated identically, thereby simplifying the implementation of later phases.

In addition to tokenization, the Gemini lexer ensures comments are balanced and that quotes are closed.

The tool ML-Lex was utilized in order to specify the lexing patterns. Each pattern is specified by a regular expression and an action, which could be reporting an error, generating a token, or side-effecting [6].

**5.2 PARSER**

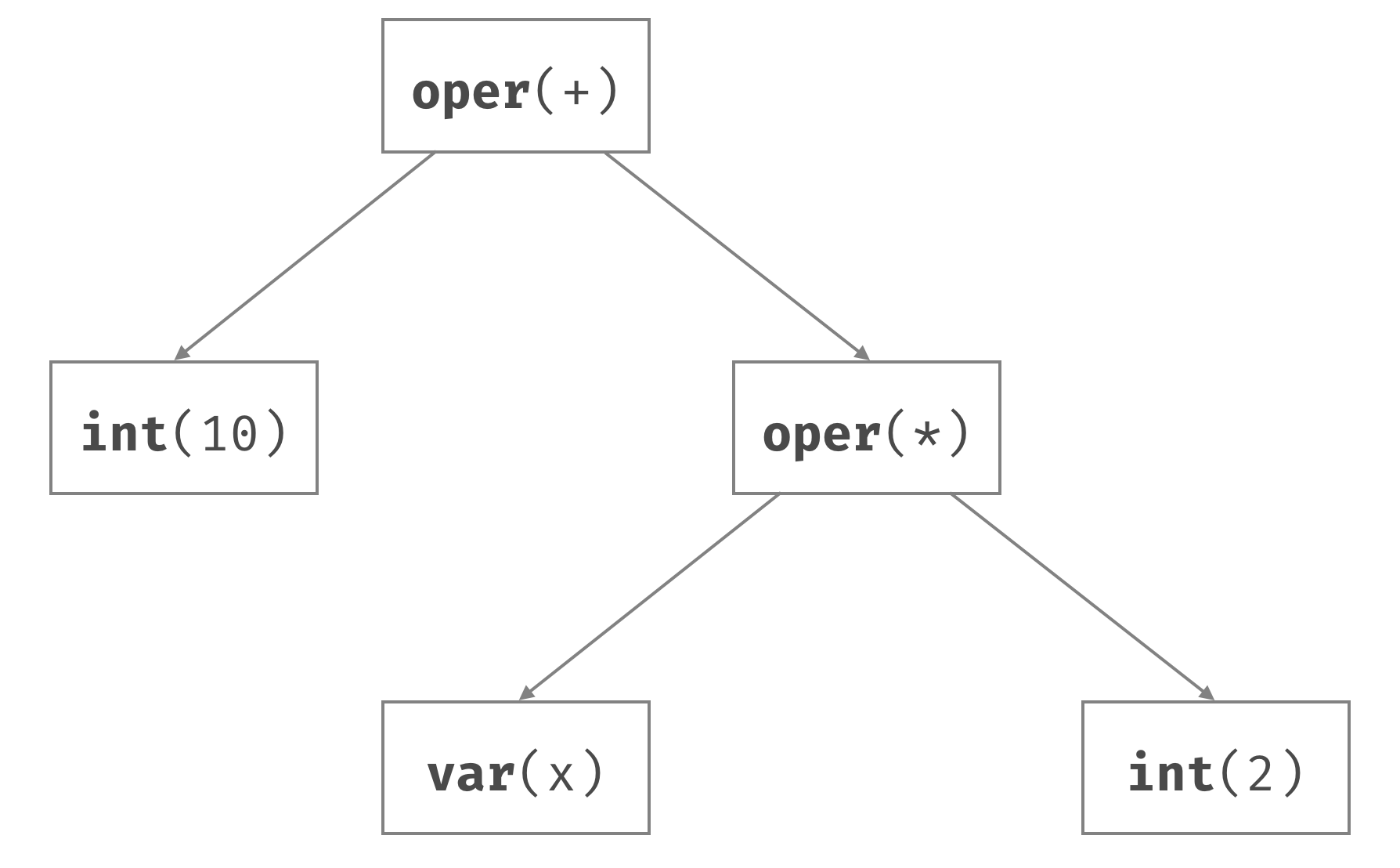
After the program passes through the lexer, we have a linear stream of tokens. In this phase, we provide structure to the tokens by constructing an *abstract syntax tree* (AST). It is *abstract* because we shed references to the concrete syntax of the language. Replacing the equality operator **=** with **==**, for example, would thereby require modification of the lexer but the AST produced by the parser would be identical.

A *context free grammar* (CFG) is used to describe all the possible strings that are grammatically valid in a given language. A CFG consists of a set of production rules; each specifies how some *non-terminal* can be replaced by a sequence of non-terminals and *terminals*, also called tokens. For example, the following grammar defines the language of strings of comma-separated s:

Production rules may be recursive, and may define multiple replacements. The special character denotes the empty string. In this grammar, the string is accepted, while the string is not.

The tool ML-Yacc was utilized to specify the CFG. A set of production rules is defined in terms of tokens provided by the lexer output and a set of non-terminals. Further, each production rule is accompanied by a semantic action to specify some return value. As the program is processed by a look-ahead LR parser, the AST is constructed from the expressions returned by the semantic actions [7]. The AST generated for the Gemini program 10 + (x \* 2) is shown below.

**Figure 5-1: Abstract syntax tree**



ML-Yacc further enables the specification of *precedence rules*, which dictate the order of precedence for terminals. The order of precedence for tokens in Gemini is shown in Figure A-8 of Appendix A.

The parser imposes grammatical correctness on the input program. This means that a lexically correct program, such as ‘**if** 10’, would be lexed successfully but rejected by the parser. It is also worthwhile to note what the parser does *not* do, which is enforce semantic validity. The Gemini program ‘42 \* “”’ is lexically and grammatically correct, although it clearly violates the typing relation and operational semantics as defined in Sections 3.4 and 3.5. This is the responsibility of the semantic analyzer to enforce.

**5.3 SEMANTIC ANALYSIS**

In this phase, colloquially known as type-checking, we recurse over the AST and verify that the semantics of the program are valid. However, a preventative issue is that not all types are currently known, since variables – which I use as a term that encompasses values, functions, and module/function parameters – may be declared implicitly. It is the responsibility of the compiler to infer the actual types, which is possible to do given that Gemini’s type system can be classified as a Hindley-Milner (HM) type system. Thus, semantic analysis is further divided into three subphases: decoration, inference, and type-checking.

In each of these phases, we will refer to the variant datatype Types.ty, shown in full in Figure D-1 of Appendix D. Formulating the type system this way allows the natural separation of kinds. There is no way, for example, to construct a function type (denoted by Types.ARROW) that accepts or produces a hardware type. Further, it is not possible to construct higher-order modules (denoted by Types.MODULE) since the constructor can only accept hardware types.

**5.3.1** Type Decoration

In the first subphase, the Gemini program is transformed into an intermediate language we will refer to as ExplicitGemini. In this language, all variables are given explicit types, as the figure below shows.

**Figure 5-2: The same program written in Gemini and ExplicitGemini**

**fun** print\_and\_mult(x, y, s : string) = (print(s); x \* y)

**fun** print\_and\_mult(x : ‘a, y : ‘b, s : string) : ‘c = (print(s); x \* y)

We introduce the notion of *type variables* which are types that can be instantiated to any type within the same kind. Type variables in Gemini are therefore different from in conventional programming languages since there is a need to differentiate between software type variables and hardware type variables. These type variables may then potentially be unified or substituted with some other type in the inference subphase. Notationally, they are written with an apostrophe followed by some alphabetic character.

In terms of the implementation, each variable in the AST possesses information about its type, which is some value of the variant datatype Absyn.ty which is shown in full in Figure D-2 of Appendix D. In the parsing phase, explicitly typed variables were given the appropriate type, such as Absyn.IntTy (for int) or Absyn.ListTy(Absyn.StringTy) (for string list) while implicitly typed variables are given Absyn.PlaceholderTy. In decoration, the AST is rebuilt with each variable having a type entry of the construtor Absyn.ExplicitTy which accepts a value of the variant datatype Types.ty. Thus, a variable that had type value Absyn.IntTy would now have type value Absyn.ExplicitTy(Types.S\_TY(T.INT)). A variable that had type value Absyn.PlaceholderTy would be explicitly given a new fresh type variable, using either the Types.S\_META or Types.H\_META constructors based on its kind.

**5.3.2** Type Inference

Type inference, also called type reconstruction, is the most complex component of the Gemini compiler. There are two primary algorithms underlying type inference: *unification* and *substitution*.

The goal of the unification algorithm is to compute the smallest possible substitution mapping from type variables to types. The unification algorithm is summarized below.

**Figure 5-3: Unification algorithm**

ISHWTYPE()

1. **case** **of**
2. H\_TY(\_) **true |** \_ **false**

ISSWTYPE()

1. **case** **of**
2. S\_TY(\_) **true** **|** \_ **false**

UNIFY()

1. **case** **of**
2. META()
3. **|** H\_META() **if** ISHWTYPE() **then** **else** **raise** KINDERROR
4. **|** S\_META() **if** ISSWTYPE() **then** **else** **raise** KINDERROR
5. | \_  **case**  **of**
6. META()
7. **|** H\_META() **if** ISHWTYPE() **then** **else** **raise** KINDERROR
8. **|** S\_META() **if** ISSWTYPE() **then** **else** **raise** KINDERROR
9. | \_  **if** ISHWTYPE() **and** ISHWTYPE()
10. **then** UNIFYHWTYPE()
11. **else** **if** ISSWTYPE() **and** ISSWTYPE()
12. **then** UNIFYSWTYPE()
13. **else** **raise** KINDERROR

The subroutines UNIFYHWTYPE and UNIFYSWTYPE are omitted for the sake of brevity. Both of these algorithms operate on the basis of structural recursion. If the two types share the same outermost type, then the appropriate unification algorithm is called on the inner types. The recursion terminates once a type variable is being unified with some other type, in which case a mapping is returned.

In substitution, the mapping returned by the unification algorithm is used to augment a global substitution environment . Since each type variable is created freshly, it is safe to maintain a global environment since no two type variables will correspond to the same element in the domain, and each type variable can only map to a single element in the domain. That is, the substitution mapping is bijective.

In addition to , there are two more environments maintained although their scope is only local to their closure. These are the type environment and the variable environment . Within a let-binding, declarations bind symbols to their types in these environments. Values, functions, and modules are bound in whereas types and datatypes are bound in . When processing a function or module, the parameters are added to the environment and only exist within the scope of the body. Since SML is a functional programming language, the augmented environments are discarded once the body has been processed, simulating how lexical scoping should act. As the AST is traversed, the mappings in are applied to both and in order to persist the results of unification. This constitutes the substitution algorithm, which is shown in pseudocode below.

**Figure 5-4: Substitution algorithm**

SUBSTITUTE(, )

1. **false**
2. **while** **do**
3. **for** (**in**  **do**
4. SUBSTITUTETYPE()
5. **return**

SUBSTITUTETYPE(,)

1. **case**  **of**
2. S\_TY() **return** SUBSW(, , , )
3. | H\_TY() **return** SUBHW (, , , )
4. | M\_TY() **return** SUBMOD (, , , )

SUBSW (,, )

1. **case** **of**
2. S\_META() **if**
3. **then return**
4. **else if**
5. **then** **case** **of**
6. S\_TY() (**case**  **of**
7. S\_META() **if**
8. **then** **true**
9. | \_ **true**;
10. **return** )
11. | \_ **return**
12. **else return**
13. | INT **return** INT
14. | ARROW(, )
15. **return** ARROW(SUBSW(), SUBSW())
16. | S\_POLY(, )
17. **return** S\_POLY(, SUBSW()
18. | S\_MU (, ) **return** S\_MU(, SUBSW()

The subroutines SUBHW and SUBMOD are omitted, but are similar to SUBSW. Some cases from SUBSW are also omitted, but the interesting ones are shown. In substituting a type variable we first determine if it is a bound variable. If it is, then we must not substitute. If it is not, we look up the mapping in and return the mapped type if it exists. We must also make sure to let the iteration algorithm know if any substitution has occurred, in order for it to continue iterating until it reaches a fixed point. Types like INT cannot be substituted any further and are returned as is. For types with inner types, such as ARROW, the SUBSW routine is called recursively. The two most interesting cases are S\_POLY and S\_MU.

The S\_POLY type is inferred any time a function is parametrically polymorphic in its arguments, and represents the mathematical idea of a universal quantifier. The set denote the type variables that are bound by the quantifier. As such, when performing substitution, these must not be substituted. Only upon function application does the S\_POLY type become instantiated, at which point each type variable in is substituted uniformly with whatever type is provided.

Before discussins S\_MU, we must first momentarily bring light to a special consideration made during the decoration phase. Since datatypes may be recursive, it is necessary to decorate their type uniquely when they are declared. The reason for this is twofold. Firstly, while processing the body of the datatype there must exist some reference to the datatype itself since the constructor may be self-referential. In decorating datatype , a temporary fresh type variable is generated and the type environment is augmented with the mapping . Then, the datatype constructors are decorated with any recursive reference to being replaced with the type variable . Once the entire datatype has been processed, the true type can determined and is augmented with the mapping . Secondly, we wish to prevent infinite substitution from occuring in the inference phase if the datatype is recursive, and as such we wrap the type

with -recursion. In substitution, whenever we encounter the operator we refrain from substituting any variables it binds. Only when constructors are instantiated do we expand the recursive definition once.

In inferring recursive functions, an approach similar to the handling of recursive datatypes is taken. Namely, the variable environment is augmented with a mapping from the function name to a function type with type variables as parameter and return types. When processing the body, any application of the recursive function can be unified since the preliminary definition was polymorphic.

Type inference enables parametric polymorphism by constraining types as loosely as possible. This allows a single part of a program to be used with different types. As an example, consider the following programs to concatenate an element to a list written in a fictional language ExplicitGemini with no type inference.

**Figure 5-5: Concatenation functions written in ExplicitGemini**

**fun** concatInt (x : int) (y : int list) : int list = x::y

**fun** concatString (x : string) (y : string list) : string list = x::y

The type of concatInt is and the type of concatString is . In Gemini, type reconstruction allows us to instead define the following.

**Figure 5-4: Concatenation function written in Gemini**

**fun** concat x y = x::y

Not only is the code less verbose, but it can be used to concatenate an element of any type to a list of the same type. The cons operator :: is parametrically polymorphic allowing it to operate on arguments of type ‘a and ‘a list. We use the universal quantifier to represent the type of the polymorphic concat as . Upon instantiation, the quantifier is removed and the type variables are substituted uniformly for some concrete type.

**5.3.3** Type-Checking

With all types inferred, it is now time to perform type-checking. The typing rules from Section 3.4 are utilized in semantic verification. This is done in a recursive manner, since propositions of typing rules make statements about the types of subexpressions in order to verify the semantics of the entire expression. The recursion terminates once a typing rule is reached that has no proposition, indicating it is an axiom.

In our implementation, type-checking and type inference are performed concurrently. This is an optimization to avoid an unnecessary additional traversal of the AST, since the typing rules can be enforced once the types of subexpressions have been inferred. This is done by augmenting the unification algorithm from Figure 5-3 to determine whether any two types can be unified, even if neither are type variables. The way that this is done is by comparing the structure of the two types. If the structures are the same, such as two arrow types, then the unification algorithm recurses on any inner types. The termination case of the recursive algorithm is when the types contains no inner types to unify, such as int and int. If the structures are different at any point, then it indicates that the two types are incompatible.

**5.4 EVALUATION**

Once semantic analysis has been performed, we can safely begin the evaluation phase. Typically, compilers do not evaluate the program they are compiling and instead produce code in some target language that is functionally equivalent to the source program. However, in the case of the Gemini compiler we wish to produce Verilog code, which does not support certain software primitives. As a result, this phase of compilation evaluates all software-typed values in order to generate an intermediate representation (IR) tree that consists solely of hardware-typed values, which is used to produce Verilog.

In our implementation, we evaluate according to the rules from Section 3.5. Having intentionally chosen small-step semantics to define our evaluation rules, translation to implementation becomes trivial. For example, below is the code that evaluates conditional expressions (modified slightly for readability).

**Figure 5-3: Evaluating conditional expressions**

evalExp(Absyn.IfExp{guard, then', else', pos}) =

let

val guardVal = evalExp(guard)

in

if (getInt(guardVal)) <> 0

then evalExp(then')

else evalExp(else’)

end

As can be seen, the guard is first evaluated recursively until a value is achieved. We retrieve the integer constant associated with the value and compare it to 0: if it is non-zero then we evaluate the then-clause, else we evaluate the else-clause. This is directly aligned with the rules E-IFTHEN, E-IFTHEN-T, and E-IFTHEN-F.

While performing evaluation, a value store is maintained that maps symbols to the values they possess, specifically utilizing the Values.value datatype shown in full in Figure D-3 of Appendix D. Any time a variable is referenced, it’s assigned value is looked up in the value store. This naturally enables lexical scoping, since the value store within an inner scope is discarded once that scope is exited.

Evaluating function and module declarations warrants special discussion. When a function is declared, its name is bound in the value store to a Values.value -> Values.value function. When this function is called, the supplied Values.value is bound to the function parameter names and the body is processed with the augmented value store. The reason this binding is made, as opposed to simply binding to the function body, is because the values of the arguments are only known upon function application. SML’s closure rules enable the function to “remember” the state of the value store upon declaration and process the body correctly after augmenting it with the parameters.

Module declarations are complicated by one more consideration. If modules are instantiated in the program itself, a similar approach can be taken in order to expand the module body inline. However, the top-level module returned is not instantiated but still needs to be expanded in order to generate Verilog. This is done by capturing the argument names when the module is declared and storing it as part of the module value, as Figure D-3 exhibits. At the top-level, the argument names are applied to the module function to expand the module body with all appearances of the argument variables replaced with Values.NamedVal, which represents some input pin in Verilog.

**5.5 VERILOG PRODUCTION**

At the final phase of compilation, we are left with a tree of hardware values representing a circuit, constituted only by bits, logical gates, arrays, and input pins. We can begin to generate the output Verilog.

The manner that this is done is not unsimilar to previous phases: we recurse over the AST and build the output from the results of subtrees. The base case of recursion is when an input pin or constant are encountered, in which case the appropriate variable name or constant declaration is returned. At each node of the hardware tree, the subexpressions are evaluated to determine their name and then the node itself is evaluated. A fresh variable name is generated and returned for superexpressions to use in computing themselves.

One final consideration is that although we performed type-checking in an earlier phase, not all types were known; recall that certain hardware types are parameterized by values, which were not evaluated until this phase. As such, an additional pass of hardware type-checking must be performed. As an optimization, this is done during the recursion of the hardware value tree instead of performing an entire traversal again.

**6 CONCLUSION**

In this project, a functional hardware description language was designed and compilation to Verilog was implemented. The final product is a tool that enables improved expressivity and modularity while preserving the benefits of the established Verilog synthesization tools

**7 REFERENCES**

[1] Barbacci, M. "A comparison of register transfer languages for describing computers and digital systems," Carnegie-Mellon Univ., Dept. of Computer Science, March 1973

[2] ["Verilog's inventor nabs EDA's Kaufman award"](http://www.eetimes.com/document.asp?doc_id=1157349). EE Times. November 7, 2005

[3] Department of Defense (1992). [Military Standard, Standard general requirements for electronic equipment](http://everyspec.com/MIL-STD/MIL-STD-0300-0499/MIL-STD-454N_9160/). Retrieved November 15, 2017

[4] Barbacci, M., Grout S., Lindstrom, G., Maloney, M.P. "Ada as a hardware description language : an initial report," Carnegie-Mellon Univ., Dept. of Computer Science, 1984

[5] Pierce, Benjamin C. Types and Programming Languages. MIT Press, 2002

[6] Appel, Andrew W., et al. “A Lexical Analyzer Generator for Standard ML.” A Lexical Analyzer Generator for Standard ML. Version 1.6.0, October 1994, Princeton University, Oct. 1994, www.smlnj.org/doc/ML-Lex/manual.html.

[7] Tarditi, David R., and Andrew W. Appel. “ML-Yacc User's Manual Version 2.3.” Princeton University, The Trustees of Princeton University, 6 Oct. 1994, www.cs.princeton.edu/~appel/modern/ml/ml-yacc/manual.html

**APPENDIX A**

**Figure A-1:** Hardware Operators

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| & | e1 & e2 | Bitwise logical “and” of composing bits\* |
| | | e1 | e2 | Bitwise logical “or” of composing bits\* |
| ^ | e1 ^ e2 | Bitwise logical “xor” of composing bits\* |
| ! | !e1 | Negation of bit operand |
| << | e1 << e2 | Shifts left bit array operand to the left by the amount specified (as an unsigned integer) by the right bit array |
| >> | e1 >> e2 | Shifts left bit array operand to the right by the amount specified (as an unsigned integer) by the right bit array, filling with zero bit value |
| >>> | e1 >>> e2 | Shifts left bit array operand to the right by the amount specified (as an unsigned integer) by the right bit array, filling with most significant bit value |
| &-> | &->e1 | Bitwise and-reduction of bit array operand |
| |-> | |->e1 | Bitwise or-reduction of bit array operand |
| ^-> | ^->e1 | Bitwise xor-reduction of bit array operand |
| && | e1 && e2 | Or-reduction of both bit array operands, followed by bitwise logical “and” of resulting bits |
| || | e1 || e2 | Or-reduction of both bit array operands, followed by bitwise logical “or” of resulting bits |
| ^^ | e1 ^^ e2 | Or-reduction of both bit array operands, followed by bitwise logical “xor” of resulting bits |

\* these operators pervade through the structures of the subexpressions to perform the bitwise operation on the composing bits while retaining the overall structure

**Figure A-2:** Arithmetic Operators

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| ~ | ~e | Negation of integer operand |
| + | e1 + e2 | Addition of both integer operands |
| - | e1 – e2 | Subtraction of right integer operand from left integer operand |
| / | e1 / e2 | Division of left integer operand by right integer operand, rounded towards negative infinity |
| \* | e1 \* e2 | Multiplication of both integer operands |
| % | e1 % e­2 | Modulo of dividend left integer operand with divisor right integer operand |
| +. | e1 +. e2 | Addition of both real operands |
| -. | e1 -. e2 | Subtraction of right real operand from left real operand |
| /. | e1 /. e2 | Division of left real operand by right real operand |
| \*. | e1 \*. e2 | Multiplication of both real operands |

**Figure A-3:** Conditional Operators

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| andalso | e1 andalso e2 | Logical conjunction of both integer operands |
| orelse | e1 orelse e2 | Logical disjunction of both integer operands |
| not | not e1 | Logical complementation of integer operand |

**Figure A-4:** Comparison Operators

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| = | e1 = e2 | Equality of both operands |
| <> | e1 <> e2 | Non-equality of both operands |
| > | e1 > e2 | Left operand has a strictly greater order than right operand |
| < | e1 < e2 | Left operand has a strictly lesser order than right operand |
| >= | e1 >= e2 | Left operand has a greater or equal order compared to right operand |
| <= | e1 <= e2 | Left operand has a lesser or equal order compared to right operand |

**Figure A-5:** List Operators

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| :: | e1::e2 | Append left element operand to right list operand |

**Figure A-6:** Types

|  |  |
| --- | --- |
| Type | Syntax |
| integer | int |
| string | string |
| real | real |
| list of TS | TS list |
| ref of TS | TS ref |
| software-wrapped TH | TH sw |
| record of TS1, …, TSn with labels l1, …, ln | {l1: TS1, …, ln: TSn} |
| tuple of TS1, …, TSn | TS1 \* … \* TSn |
| function from TS1 to TS2 | TS1 -> TS2 |
| bit | bit |
| array of TH | TH[n] where n is some integer literal |
| temporal of TH | TH @ n where n is some integeral literal |
| record of TH1, …, THn with labels l1, …, ln | #{l1: TH1, …, ln: THn} |
| tuple of TH1, …, THn | TH1 #\* … #\* THn |

**Figure A-7:** Library Functions

|  |  |  |  |
| --- | --- | --- | --- |
| Structure | Function | Type | Semantic Result |
| Core | print | string -> unit | Write a string to the standard output |
| read | string -> string | Read the contents of a file |
| List | nth | (‘a list \* int) -> ‘a | Return an element from a list given an index; raises an exception if the index is out of bounds |
| length | ‘a list -> int | Return the length of a list |
| rev | ‘a list -> ‘a list | Return the reversed list |
| map | (‘a -> ‘b) -> ‘a list -> ‘b list | Return a list after applying a function to each element |
| filter | (‘a -> int) -> ‘a list -> ‘a list | Return a list containing only elements that satisfy the predicate function |
| foldl | (‘a \* ‘b -> ‘b) -> ‘b -> ‘a list -> ‘b | Accumulate a value by iterating over a list from left to right |
| foldr | (‘a \* ‘b -> ‘b) -> ‘b -> ‘a list -> ‘b | Same as foldl except iteration is right to left |
| Int | toString | int -> string | Return a string representation of an int |
| String | size | string -> int | Return the number of characters in a string |
| substring | (string \* int \* int) -> string | Return the substring from a start to end location of a string; raises an exception if either index is out of bounds |
| concat | string list -> string | Return the concatenation of all strings in a list |
| split | string -> string -> string list | Return a list of strings resulting from splitting an original string over some delimiter |
| Real | floor | real -> int | Return a real rounded towards negative infinity |
| ceil | real -> int | Return a real rounded towards positive infinity |
| round | real -> int | Return a real rounded towards the closest integer |
| fromInt | int -> real | Return a real converted from an integer |
| toString | real -> string | Return a string representation of a real |
| Array | toList | ‘a[n] sw -> ‘a sw list | Return a list of software-wrapped hardware values from a software-wrapped hardware array |
| fromList | ‘a sw list -> ‘a[n] sw | Return a software-wrapped hardware array from a list of software-wrapped hardware values |
| BitArray | twosComp | bit[n] ~> bit[n] | Return a circuit performing twos-complement of a bit array |
| HW | dff | ‘a ~> ‘a @ 1 | Return a DFF circuit with a given hardware input |

**Figure A-8:** Operator order of precedence, from high to low

|  |  |
| --- | --- |
| Operator(s) | Associativity |
| ~, !, |->, &->, ^-> | N/A |
| $ | N/A |
| #f | N/A |
| [:i:] | N/A |
| /., \*., /, \*, &, % | left |
| -., +., -, +, ^, | | left |
| && | left |
| ||, ^^ | left |
| :: | right |
| >, <, >=, <= | left |
| =, <> | left |
| <<, >>, >>> | left |
| andalso | left |
| orelse | left |
| := | right |

**APPENDIX B**

**Figure B-1: Derived Terms**

|  |  |
| --- | --- |
| Name | Equivalence |
| tuple |  |
| unit | () {} |
| logical and | e1 andalso e2 if e1 then e2 else 0 |
| logical or | e1 orelse e2 if e1 then 1 else e2 |
| logical not | not e1 if e1 then 0 else 1 |
| and-reduction | &->#[e1, e2] e1 & e2 |
| or-reduction | |->#[e1, e2] e1 | e2 |
| xor-reduction | ^->#[e1, e2] e1 ^ e2 |
| and collapse | e1 && e2 (|->e1) & (|->e­2) |
| or collapse | e1 || e2 (|->e1) | (|->e­2) |
| xor collapse | e1 ^^ e2 (|->e1) ^ (|->e­2) |
| if-then | if e1 then e2 if e1 then e2 else {} |
| sequence | (e1; e2) (T.t2) t1 where t2 |

**APPENDIX C**

**APPENDIX D**

**Figure D-1: Types.ty datatype**

**datatype** ty = H\_TY **of** h\_ty

| S\_TY **of** s\_ty

| M\_TY **of** m\_ty

| META **of** tyvar

| TOP

| BOTTOM

**and** h\_ty = BIT

| ARRAY **of** {ty: h\_ty, size: **int** **ref**}

| TEMPORAL **of** {ty: h\_ty, time: **int** **ref**}

| H\_RECORD **of** (tyvar \* h\_ty) **list**

| H\_DATATYPE **of** (tyvar \* h\_ty **option**) **list** \* **unit ref**

| H\_POLY **of** tyvar **list** \* h\_ty

| H\_META **of** tyvar

| H\_TOP

| H\_BOTTOM

**and** s\_ty = INT | REAL | STRING

| ARROW **of** (s\_ty \* s\_ty)

| LIST **of** s\_ty

| SW **of** h\_ty

| S\_RECORD **of** (tyvar \* s\_ty) **list**

| REF **of** s\_ty

| S\_DATATYPE of (tyvar \* s\_ty **option**) **list** \* **unit** **ref**

| S\_MU **of** tyvar **list** \* s\_ty

| S\_POLY **of** tyvar **list** \* s\_ty

| S\_META **of** tyvar

| S\_TOP

| S\_BOTTOM

**and** m\_ty = MODULE **of** h\_ty \* h\_ty

| PARAMETERIZED\_MODULE **of** s\_ty \* h\_ty \* h\_ty

| M\_POLY **of** tyvar **list** \* m\_ty

| M\_BOTTOM

**Figure D-2: Absyn.ty datatype**

**datatype** ty = NameTy **of** symbol \* pos

|ParameterizedTy **of** symbol **\*** (ty **list**) **\*** pos

|TyVar **of** symbol **\*** pos

|SWRecordTy **of** field **list \*** pos

|HWRecordTy **of** field **list \*** pos

|ArrayTy **of** ty **\* int \*** pos

|ListTy **of** ty **\*** pos

|TemporalTy **of** ty **\* int \*** pos

|RefTy **of** ty **\*** pos

|SWTy **of** ty **\*** pos

|FunTy **of** ty **\*** ty **\*** pos

|PlaceholderTy **of unit ref**

|ExplicitTy **of** Types.ty