Synthesizable Verilog[*]

Cherif Andraos Jennifer Gillenwater Gregory Malecha Angela Yun Zhu
Walid Taha Jim Grundy John O'Leary

Abstract

To ensure that hardware descriptions are synthesizable, designers today bear the responsibility of understanding and conforming to coding guidelines. This makes it harder to write reusable designs, and makes the transition from a behavioral design to a structural design more labor intensive. This paper proposes the use of a static analysis to check the synthesizability of a design. This static analysis uses ideas from statically-typed two-level languages to reflect the distinction between values that are known at design time and values that are carried on wires in synthesized circuits. This distinction is crucial for determining whether abstractions such as iteration and module parameters are used in a synthesizable manner.

To formalize these ideas, we develop a core calculus for Verilog, along with a static type system. We define a preprocessing step that defines how such abstractions can be eliminated in an early phase of synthesis, and the kinds of errors that can occur during preprocessing. Our key technical result is showing that a well-typed design cannot lead to preprocessing errors, and can only expand to a synthesizable circuit.

1 Introduction

Verilog is a hardware description language aimed at both simulation and synthesis of digital circuits. All descriptions can be simulated, but only some can be synthesized. Unfortunately, Verilog does not clearly reflect the boundary between what can and what cannot be synthesized. In fact, what is synthesizable can depend on the particular synthesis tool being used. To ensure that their designs can be synthesized, designers use semi-formal guidelines when writing hardware descriptions. Because designers bear the responsibility of enforcing these guidelines, they tend to err on the side of safety. In doing so, they forgo significant opportunities for using expressive abstraction mechanisms such as iteration and module parameterization. For example, using `for`-loops and module parameterization, the design of a simple encoder can be described concisely and generically as follows:

```
module encode (L,X);
  parameter n = 2;     parameter m = 4; // 2^n
```

[*]This work was supported by the National Science Foundation (NSF) and the Semiconductor Research Consortium (SRC).

```
  input  [m-1:0] L;    output [n-1:0] X;
  reg    [n-1:0] X;    integer        i;

  always @(L)
    for (i=0; i<m; i=i+1) if(L[i]==1) X=i;
endmodule
```

Unfortunately, in both educational examples and in industrial designs, we often find that the main computation in the `for`-loop is replaced by the much more specialized code:

```
      if(L[0]==1) X=0; if(L[1]==1) X=1; if(L[2]==1) X=2; if(L[3]==1) X=3;
```

The result of such practice is reduced readability and increased redundancy among related designs. Whereas the generic version above can be used to synthesize several encoders of different sizes, manually producing several specialized versions makes the design base unnecessarily hard to produce and maintain.

Our thesis is that the process of going from generic designs to specialized ones is highly mechanizable, and should be formalized and treated as an integral part of the specification of hardware description languages such as Verilog and VHDL. In particular, if designers can *rely* on a synthesis tool to perform this process, they can avoid having to do this manually. Two things must be defined to achieve this goal: First, a static analysis that subsumes the semi-formal guidelines, and that automatically verifies that all abstraction mechanisms used in a design are "safe for synthesis". Second, a clear specification of the abstraction process which can be used as a basis for specifying what is required from a synthesis tool. As we will show in this paper, statically-typed two-level languages (STTLs) [3] that have been used to model statically-typed macros and statically-typed program generators [6, 5, 2] can provide the formal infrastructure needed to achieve these goals.

1.1 Contributions

We begin by defining a representative calculus to model Verilog descriptions. This requires identifying the core concepts reflected in Verilog's type system, as well as determining what can be left out without limiting the practical applicability of the formal treatment (Section 2). Using ideas from two-level languages, we define a type system for this calculus. This type system reflects – at a very high-level – the conditions needed to guarantee that various abstractions do not interfere with synthesis (Section 3). Next, we develop an operational two-level semantics for the calculus. This semantics captures both how various abstraction mechanisms should be preprocessed during synthesis, as well as what can go wrong in this process (Section 4).

We establish three key properties for this design (Section 5). First, a well-typed design synthesizes to a well-typed, one-level Verilog description (Theorem 5.1). Second, preprocessing is safe in the sense that none of its actions depend on wire values (Theorem 5.2). Third, the synthesized one-level description is free of preprocessing (or "stage one") computations (Theorem 5.3). Together, these properties show that the result of expanding any well-typed description is a trivially synthesizable circuit.

2 Core Calculus

The calculus models the core abstractions of Verilog, which are *signals* and *modules*. Intuitively, a signal is a wire or an array of wires. A module is a circuit with input and output signals, as well as internal state in the form of *registers*. In addition, a module's type (and behavior) can be parameterized by a set of integer-valued *parameters*.

Verilog provides many constructs to make writing hardware descriptions as convenient as possible. In contrast, a calculus must be minimized to facilitate analysis of the essential features of the language. With this goal in mind, we model only signals, registers, iterations, and modules. We leave out other features of Verilog, such as being able to connect modules' ports by name, default parameter values, initialization blocks, fork/join, repeat, and forever.

2.1 Notational Simplifications

The calculus will have a form that closely resembles the concrete syntax for Verilog, but is not exactly the same. The syntax has been simplified for notational convenience. In particular:

- All sequences are represented uniformly as `<a,b,...,z>`,
- Parentheses of a module body are replaced by a terminal `is` to delineate the header from the body of the module,
- Local variable declarations are aggregated immediately after the `is` terminal,
- Module parameters are not declared local, but rather, listed separately before the usual formal parameters,
- The type and direction of the formal argument list to a module are declared in the formal argument itself, rather than in the body of the module definition,
- Names of several Verilog keywords, such as `input`, `output`, and `register`, are replaced by shorter names, such as `in`, `out`, and `reg`,
- Direction of variables is represented using a set that can be equal to $\{\texttt{in}\}$, $\{\texttt{out}\}$, $\{\texttt{in}, \texttt{out}\}$, or $\emptyset$ denoting `in`, `out`, `inout`, and no direction respectively. The later is used for non-directional variables like parameters for example.
- `If`-statements always have an `else` clause, since the language allows empty sequences of statements,
- Because Verilog uses automatic coercions to pad arrays of wires of different size to match them, wire sizes are dropped from terms,
- `For`-loop variables do not need to be declared explicitly and are local to the `for`-loop, and
- Integers are represented in binary to abstract from the parsing process.

Using these conventions, the generic version of the encoder presented in the introduction can be expressed as follows:

```
< module encode <n,m> <in wire L, out reg X> is <>
    < always @(L)
       for(i = 0; i < m; i=i+1 )
         if(L[i] == 1) then X = i else <>
    >,
  module main <> <in wire L, out reg X> is <>
    < encode <2,4> <L,X>
    >,
> main
```

In the rest of the paper, we will also use the following conventions in the formal treatment of the calculus.

Notational Conventions 2.1.

- *A sequence of elements drawn from the set X is either the empty sequence $\langle\rangle$ or a non-empty sequence $h :: t$ with a head $h \in X$ and a tail sequence t of elements drawn from the same set.*
- *We write $\langle X_i \rangle^{i \in I}$ to denote a sequence of elements drawn from the set X. The index set I is a subset of the naturals. The length of the sequence is the same as the size of I.*
- *When it is clear from context, we will drop the index set and write $\langle X_i \rangle$ instead of $\langle X_i \rangle^{i \in I}$.*
- *We write $X \uplus Y$ for the concatenation of the two sequences X and Y.*
- *We write $\biguplus_{k} \langle D_r \rangle^{r \in R(k)}$ for the concatenation of all $\langle D_r \rangle^{r \in R(k)}$.*
- *We write $|\langle X_i \rangle|$ for the length of the sequence $\langle X_i \rangle$.*
- *X^* indicates the set of lists of X terms.*
- *X^+ indicates the set of non-empty lists of X terms.*

2.2 Formal Syntax (BNF)

The abstract syntax for the calculus is parameterized by sets for identifier and operator names. In addition, it will be convenient to use several meta-variables to range over indices and index domains.

Module	m	$\in$	ModuleNames, a countably infinite set
Signal or Level 1 Local Variable	s	$\in$	IdentifierNames, a countably infinite set
Parameter or Level 0 Local Variable	x, y	$\in$	ParameterNames, a countably infinite set
Operator	$\mathbb{O} \ni f$	$\in$	OperatorNames, a finite set
Index	h, i, j, k, q, r	$\in$	$\mathbb{N}$, the set of natural numbers
Index Domain	H, I, J, K, Q, R	$\subseteq$	$\mathbb{N}$

The full grammar for the calculus is defined by the following Backus-Naur Form (BNF):

$$
\begin{array}{llcl}
\textit{Program} & p & ::= & \langle D_i\rangle^{i\in I}\ m \\
\textit{Module Definition} & D & ::= & \texttt{module}\ m\ b \\
\textit{Module Body} & b & ::= & \langle x_i\rangle^{i\in I}\ \langle d_j\ t_j\ s_j\rangle^{j\in J}\ \texttt{is}\ \langle t_k\ s_k\rangle^{k\in K}\ \langle P_r\rangle^{r\in R} \\
\textit{Direction} & d & \subseteq & \{\texttt{in},\ \texttt{out}\} \\
\textit{Type} & \mathbb{T} \ni t & ::= & \texttt{wire} \mid \texttt{reg} \mid \texttt{int} \\
\textit{Parallel Statement} & P & ::= & m\ \langle e_i\rangle^{i\in I}\ \langle l_j\rangle^{j\in J} \mid \texttt{assign}\ l\ e \mid \texttt{always}\ S \\
\textit{LHS value} & l & ::= & s \mid s[e] \mid s[e\ :\ e] \\
\textit{Sequential Statement} & S & ::= & @E^{+}\ S \mid l = e \mid \texttt{if}\ e\ \texttt{then}\ S\ \texttt{else}\ S \\
 & & & \mid \texttt{for}(y = e; e; y = e)S \mid S^{*} \\
\textit{Event} & E & ::= & g\ l \\
\textit{Edge} & g & ::= & \texttt{posedge} \mid \texttt{negedge} \mid \texttt{edge} \\
\textit{Expression} & e & ::= & l \mid x \mid v \mid f\langle e_i\rangle^{i\in I} \\
\textit{Value} & v & ::= & (0 \mid 1)^{32}
\end{array}
$$

A program $\langle D_i\rangle^{i\in I}\ m$ is a sequence of module definitions $\langle D_i\rangle^{i\in I}$ followed by a module name m. The module name indicates which module from the preceding sequence contains the overall input and output of the system. A module definition is a name and a module body. A module body itself consists of a sequence of module parameter names, a sequence of port declarations (carrying direction, type, and name for each port), a sequence of local variable declarations, and a sequence of parallel statements. Ports can have either `in`, `out`, or `inout` direction. The type of a port or a local variable can be wire, register, or integer. A parallel statement can be a module instantiation, an assign statement, or an always statement carrying a sequential statement. A module instantiation provides module parameters as well as what gets connected to various ports. An assign statement consists of a left hand side (LHS) value and an expression. An LHS value is either a variable, an array lookup, or an array range. A sequential statement is either a guarded statement, an assignment, a conditional statement, a `for`-loop, or a sequence of sequential statements. An event consists of an edge trigger (positive, negative, or either) and an LHS value. An expression is either an LHS value, a parameter name, an integral value, or an operator application. Values are 32-bit integers.

3 Type System

The type system presented in this section specifies what is a trivially synthesizable description. It also specified what preprocessing computations are implicitly embodied in a design that uses abstraction mechanisms such as `for`-loops and module parameters. In the terminology of two-level languages, preprocessing is the level 0 computation, and the remaining computation is considered to be the level 1 part. In a Verilog description, there are relatively few places where level 0 computations are required. In our calculus, these places are restricted to: 1) expressions that relate to module parameters, 2) expressions that relate to the bounds on a `for`-loop, and 3) indices into arrays.

For notational convenience, by default, the typing judgment (generally of the form $\Delta \vdash X$) will be assumed to be a level 1 judgment. That is, it is checking for validity of a description as a computation that has already been pre-processed. Expressions, however, may be computations that either must be performed during expansion or that must remain intact to become part of the result of preprocessing. For this reason, the judgment for expressions will be annotated with a level $n \in \{0, 1\}$ to indicate we are checking this expression for validity at level 0 or at level 1. This annotation will appear in the judgment as a superscript on the turnstyle as shown in $\vdash^n$.

The type system presented here will not address termination restrictions on synthesizable `for`-loops. While many synthesis tools use such restriction to ensure that synthesis terminates, they are orthogonal to the problem addressed by a type system, and so should be addressed separately.

3.1 Typing Environments

To define the type system we need some auxiliary notions. A module type consists of the number of its module parameters, as well as a list of directions and types for ports. An operator signature is a function that takes an operator, the level at which the operation is executed, the types of the operands and returns the type of the result. As noted above, levels can be 0 or 1. A module environment associates names of modules with their corresponding types while a variable environment associates variable names with their corresponding directions and types. We do not have to keep level information in the variable environment because we can differentiate between levels syntatically. All signals and declarable local variables (denoted by s) are considered level 1 variables while parameters and `for`-loop variables (denoted by x or y) are considered level 0 variables.

$$\begin{array}{llll}
\textit{Module Type} & M & ::= & k\ \langle d_i\, t_i\rangle^{i\in I} \\
\textit{Operator Signatures} & \Sigma & \in & \Pi i.\ \mathbb{O}\times\mathbb{N}\times\mathbb{T}^i \to \mathbb{T} \\
 & & & \\
\textit{Level} & n & ::= & 0 \mid 1 \\
\textit{Module Environment} & \Delta & ::= & [\,] \mid m : M\ ::\ \Delta \\
\textit{Variable Environment} & \Gamma & ::= & [\,] \mid s : d\,t\ ::\ \Gamma \mid x : d\,t\ ::\ \Gamma \\
\textit{Level 1 Variable Environment} & \Gamma^+ & ::= & [\,] \mid s : d\,t\ ::\ \Gamma^+
\end{array}$$

3.2 Well-formed Directions

In Verilog, directions only make sense as a property of signals, not parameters. Directions play a necessary role in determining what values can be used in expressions, and what values are assignable. In the calculus and type system, we prefer to address this distinction at a higher level. To this end, we will use the following judgment to specify when the direction of a value allows us to treat it as "readable":

$$\boxed{\vdash^n d}$$

$$\vdash^0 \emptyset \qquad \vdash^1 \{\texttt{in}\}$$

3.3 Typing Rules

A description is well-typed when the judgment $\vdash p$ defined in Figure 1 is derivable. A program p is typable when the declarations it contains produce a valid module environment and the main module has a type that involves 0 module parameters. A sequence of declarations is essentially a module type environment transformer: Each declaration in the sequence extends the module type environment with new declarations. To type the body of a module, we check that the body of the module is typable in the context of the current environment extended with the formal parameters as well as the local variables. It is worth noting here that: 1) local variables are viewed as `inout` signals, and 2) variables are added to form Γ without specifying their levels which are understood implicitly to be level 0 for all parameters $\langle x_i \rangle$ and level 1 for all signals $\langle s_j \rangle$ and local variables $\langle s_k \rangle$.

The next interesting rule is the one for module instantiation, which requires that the expressions relating to module parameters must be typable as level 0 computations. In contrast, the rule for `assign` requires that both LHS and RHS expressions are typable at level 1 since the assignment is a computation performed by the synthesized circuit, and not before. Similar constraints are used in (T-SAassign) and (T-If). The rule for `assign` is somewhat peculiar, because it does not require that t_1 and t_2 are the same. The only type in Verilog is `wire`, and because of the padding semantics of wires of different sizes, the Verilog type system does not really enforce that wire sizes match.

LHS expressions are merged into the rules for expressions to avoid notational redundancy.

The rule for `for`-loop requires that initialization, test, and increment expressions are all typable at level 0. The test and increment expressions require that the environment be extended to include the counter variable as being an integer (with direction $\emptyset$). If any of these expressions is not typable at level 0, the `for`-loop is rejected by the type system.

The rules for expressions are the most intricate. They only allow LHS values to be typed at level 1. In case of T-Index and T-Range, the rules also require that the indices be typable at level 0 as being integers. The rules for T-Param, T-Int and T-Op specify the direction of the expression under consideration according to the level we are typing it at. Note that the rule for operators (T-Op) implicitly requires that the operator name as well as the associated typing can be found in Σ.

4 Operational Semantics for Synthesis

A big-step operational semantics indexed by the level of the computation will be used to formally specify what preprocessing must be done. The specification will dictate how expansion should be performed, what the form of the preprocessed program should be, and what errors can occur during preprocessing.

4.1 Substitution

The key to specifying hygienic preprocessing is to use a notion of substitution that respects the binding structure of variables in programs. We achieve this by defining a standard notion of

$\boxed{\vdash p}$

$$\frac{\vdash \langle D_i\rangle : \Delta \qquad \Delta(m) = 0\,\langle d_i\; t_i\rangle}{\vdash \langle D_i\rangle\; m}\ (\mathrm{T{-}Prog})$$

$\boxed{\Delta \vdash \langle D_i\rangle : \Delta}$

$$\frac{}{\Delta \vdash \langle\rangle : [\,]}\ (\mathrm{T{-}MEmpty}) \qquad \frac{\Delta \vdash b : M \qquad m : M :: \Delta \vdash \langle D_i\rangle : \Delta'}{\Delta \vdash \texttt{module}\; m\; b :: \langle D_i\rangle \;:\; m : M :: \Delta'}\ (\mathrm{T{-}MSeq})$$

$\boxed{\Delta \vdash b : M}$

$$\frac{\{d_j \neq \emptyset\} \qquad \{t_j \neq \texttt{int}\} \qquad \{\Delta; \langle x_i : \emptyset\;\texttt{int}\rangle \uplus \langle s_j : d_j\; t_j\rangle \uplus \langle s_k : \{\texttt{in}, \texttt{out}\}\; t_k\rangle \vdash P_r\}}{\Delta \vdash \langle x_i\rangle\; \langle d_j\; t_j\; s_j\rangle\; \texttt{is}\; \langle t_k\; s_k\rangle\; \langle P_r\rangle : |\,\langle x_i\rangle\,|\langle d_j\; t_j\rangle}\ (\mathrm{T{-}Body})$$

$\boxed{\Delta; \Gamma \vdash P}$

$$\frac{\begin{array}{l} \{\Gamma \vdash^0 e_i : d_i\;\texttt{int} \qquad \texttt{in} \in d_i\} \\ \Delta(m) = |\,\langle e_i\rangle\,|\,\langle d_j t_j\rangle \\ \{\Gamma \vdash^1 l_j : d'_j t_j\} \\ \{d_j \subseteq d'_j\} \end{array}}{\Delta; \Gamma \vdash m\langle e_i\rangle\; \langle l_j\rangle}\ (\mathrm{T{-}ModInst}) \quad \frac{\begin{array}{l} \texttt{out} \in d_1 \\ \Gamma \vdash^1 l : d_1\; t_1 \\ \texttt{in} \in d_2 \\ \Gamma \vdash^1 e : d_2\; t_2 \end{array}}{\Delta; \Gamma \vdash \texttt{assign}\; l\; e}\ (\mathrm{T{-}Assign}) \quad \frac{\Gamma \vdash S}{\Delta; \Gamma \vdash \texttt{always}\; S}\ (\mathrm{T{-}Always})$$

$\boxed{\Gamma \vdash^n l : d\; t}$ See $\Gamma \vdash^n e : d\; t$

$\boxed{\Gamma \vdash S}$

$$\frac{\begin{array}{l} \{\Gamma \vdash E_i\} \\ \Gamma \vdash S \end{array}}{\Gamma \vdash @\langle E_i\rangle\; S}\ (\mathrm{T{-}Trigger}) \quad \frac{\begin{array}{l} \texttt{out} \in d_1 \\ \Gamma \vdash^1 l : d_1\; \texttt{reg} \\ \texttt{in} \in d_2 \\ \Gamma \vdash^1 e : d_2\; t \end{array}}{\Gamma \vdash l \;=\; e}\ (\mathrm{T{-}SAssign}) \quad \frac{\begin{array}{l} \texttt{in} \in d \\ \Gamma \vdash^1 e : d\; \texttt{int} \\ \Gamma \vdash S_1 \\ \Gamma \vdash S_2 \end{array}}{\Gamma \vdash \texttt{if}\; e\; \texttt{then}\; S_1\; \texttt{else}\; S_2}\ (\mathrm{T{-}If})$$

$$\frac{\begin{array}{l} \Gamma, y : \emptyset\;\texttt{int} \vdash S \\ \Gamma, y : \emptyset\;\texttt{int} \vdash^0 e_2, e_3 : \emptyset\;\texttt{int} \\ \Gamma \vdash^0 e_1 : \emptyset\;\texttt{int} \end{array}}{\Gamma \vdash \texttt{for}(y = e_1; e_2; y = e_3) S}\ (\mathrm{T{-}For}) \quad \frac{\{\Gamma \vdash S_i\}}{\Gamma \vdash \langle S_i\rangle}\ (\mathrm{T{-}SSeq})$$

$\boxed{\Gamma \vdash E}$

$$\frac{\Gamma \vdash^1 l : d\; t \quad \texttt{in} \in d}{\Gamma \vdash g\; l}\ (\mathrm{T{-}Event})$$

$\boxed{\Gamma \vdash^n e : d\; t}$

$$\frac{\Gamma(s) = d\; t}{\Gamma \vdash^1 s \;:\; d\; t}\ (\mathrm{T{-}Id}) \quad \frac{\begin{array}{l} \Gamma(s) = d\; t \\ \Gamma \vdash^0 e : \emptyset\;\texttt{int} \end{array}}{\Gamma \vdash^1 s[e] : d\; t}\ (\mathrm{T{-}Index}) \quad \frac{\begin{array}{l} \Gamma(s) = d\; t \\ \Gamma \vdash^0 e_1, e_2 : \emptyset\;\texttt{int} \end{array}}{\Gamma \vdash^1 s[e_1 \;:\; e_2] : d\; t}\ (\mathrm{T{-}Range})$$

$$\frac{\begin{array}{l} \vdash^n d \\ \Gamma(x) = \emptyset\;\texttt{int} \end{array}}{\Gamma \vdash^n x \;:\; d\;\texttt{int}}\ (\mathrm{T{-}Param}) \quad \frac{\vdash^n d}{\Gamma \vdash^n v : d\;\texttt{int}}\ (\mathrm{T{-}Int}) \quad \frac{\{\Gamma \vdash^n e_i : d\; t_i\} \qquad \vdash^n d}{\Gamma \vdash^n f\langle e_i\rangle : d\; \Sigma|\langle e_i\rangle|(f, n, \langle t_i\rangle)}\ (\mathrm{T{-}Op})$$

Figure 1: Type System

substitution using the usual free and bound variable conventions as in the lambda calculus [1].

$\boxed{P[x \mapsto v]}$

$$\begin{array}{lcl}
m\langle e_i\rangle\langle l_i\rangle[x \mapsto v] & = & m\langle e_i[x \mapsto v]\rangle\langle l_i[x \mapsto v]\rangle \\
(\texttt{assign}\ l\ e)[x \mapsto v] & = & \texttt{assign}\ l[x \mapsto v]\ e[x \mapsto v] \\
(\texttt{always}\ S\ )[x \mapsto v] & = & \texttt{always}\ S[x \mapsto v]
\end{array}$$

$\boxed{l[x \mapsto v]}$ See $e[x \mapsto v]$

$\boxed{S[x \mapsto v]}$

$$\begin{array}{lcl}
(@\ \langle E\rangle\ S)[x \mapsto v] & = & @\ \langle E[x \mapsto v]\rangle S[x \mapsto v] \\
(l = e)[x \mapsto v] & = & l[x \mapsto v] = e[x \mapsto v] \\
(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2)[x \mapsto v] & = & \texttt{if}\ e[x \mapsto v]\ \texttt{then}\ S_1[x \mapsto v]\ \texttt{else}\ S_2[x \mapsto v] \\
(\texttt{for}(y = e_1;\ e_2;\ y = e_3)\ S)[x \mapsto v] & = & \texttt{for}\begin{pmatrix} y = e_1[x \mapsto v]; \\ e_2[x \mapsto v]; \\ y = e_3[x \mapsto v] \end{pmatrix} S[x \mapsto v] \\
\langle S_i\rangle[x \mapsto v] & = & \langle S_i[x \mapsto v]\rangle
\end{array}$$

$\boxed{E[x \mapsto v]}$

$$\begin{array}{lcl}
(g\ l)[x \mapsto v] & = & g\ l[x \mapsto v]
\end{array}$$

$\boxed{e[x \mapsto v]}$

$$\begin{array}{lcl}
s[x \mapsto v] & = & s \\
s[e][x \mapsto v] & = & s[e[x \mapsto v]] \\
s[e_1 : e_2][x \mapsto v] & = & s[e_1[x \mapsto v] : e_2[x \mapsto v]] \\
x[x \mapsto v] & = & v \\
y[x \mapsto v] & = & y \qquad \text{if } y \neq x \\
v'[x \mapsto v] & = & v' \\
f\langle e_i\rangle[x \mapsto v] & = & f\langle e_i[x \mapsto v]\rangle
\end{array}$$

4.2 The Preprocessing Component of Synthesis

To model the possibility of error during preprocessing, we define the following two auxilliary notions:

$$\begin{array}{lll}
\textit{Term} & X & ::= \ p \mid D \mid b \mid P \mid l \mid S \mid E \mid e \\
\textit{Possible Term} & X_\perp & ::= \ X \mid \texttt{error}
\end{array}$$

This allows us to write $p_\perp$ or $E_\perp$ to denote a value that may either be the constant `error` or a value from p or E, respectively.

Preprocessing will be defined by the derivability of judgments of the general form $\langle D_i\rangle \vdash X \overset{n}{\hookrightarrow} X_\perp$. When the $\langle D_i\rangle$ component is irrelevant to the judgment, it will simply be dropped. Before we present the rules defining these judgments, we summarize the key points in the behavior of preprocessing. Preprocessing takes a program and starts by instantiating the main module. Other modules are instantiated as needed. At this level, the only error that can occur is that the main module may not be defined in the rest of the program. In addition, all steps of preprocessing must propagate any errors that arise in sub-computations.

Preprocessing the body of the main module involves preprocessing each of its statements in sequence. Processing each statement can involve instantiating several modules, and all such modules are aggregated (in order) and returned as the result of processing the main module body. An error occurs if the list of parameters to this module is non-empty.

Preprocessing a module instantiation generates a new module representing a unique instance of the module definition. An error can occur if the module is not defined in the context, or if the number of arguments used to instatiate a module does not match the number required by its definition.

Preprocessing the assignment statement is trivial because all the work is done ahead of time by substitution. The rules for `if` are relatively straightforward.

Preprocessing a `for`-loop essentially amounts to evaluating a `for`-loop, except that the result of evaluation is a sequence of statements rather than a modification of the global state.

Expressions need level 1 and level 0 processing. For expressions at level 0 the only active rule in evaluation pertains to operator applications. However, it is important that the semantics is explicit about the kinds of errors that can occur during evaluation, and in particular that encountering any identifier during level 0 evaluation constitutes a runtime error. This formalizes the property that any dependency on either an uninstantiated parameter or a wire value constitutes a preprocessing error.

$\boxed{p \overset{1}{\hookrightarrow} p_\perp}$

$$\frac{\texttt{module}\ m\ b \in \langle D_i\rangle \qquad \langle D_i\rangle \vdash b \overset{1}{\hookrightarrow} b', \langle D_r\rangle}{\langle D_i\rangle\ m \overset{1}{\hookrightarrow} \langle D_r\rangle \uplus \langle \texttt{module}\ m\ b'\rangle\ m}\ (\text{E}-\text{Prog})$$

$$\frac{\texttt{module}\ m\ b \notin \langle D_i\rangle}{\langle D_i\rangle\ m \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{PErr1}) \qquad \frac{\texttt{module}\ m\ b \in \langle D_i\rangle \qquad \langle D_i\rangle \vdash b \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}{\langle D_i\rangle\ m \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{PErr2})$$

$\boxed{\langle D_i\rangle \vdash b \overset{1}{\hookrightarrow} b_\perp, \langle D_i\rangle}$

$$\frac{\{\langle D_i\rangle \vdash P_k \overset{1}{\hookrightarrow} P'_k, \langle D_r\rangle^{r\in R(k)}\}}{\langle D_i\rangle \vdash \langle\rangle\langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\langle P_k\rangle \overset{1}{\hookrightarrow} \langle\rangle\langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\langle P'_k\rangle, \biguplus_k \langle D_r\rangle^{r\in R(k)}}\ (\text{E}-\text{Body})$$

$$\frac{I \neq \emptyset}{\langle D_i\rangle \vdash \langle x_i\rangle^{i\in I}\langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\langle P_k\rangle \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}\ (\text{E–BErr1}) \qquad \frac{\exists k.\langle D_i\rangle \vdash P_k \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}{\langle D_i\rangle \vdash \langle\rangle\langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\langle P_k\rangle \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}\ (\text{E–BErr2})$$

$\boxed{\langle D_i\rangle \vdash P \overset{1}{\hookrightarrow} P_\perp, \langle D_i\rangle}$

$$\frac{\begin{array}{l}\{e_i \overset{0}{\hookrightarrow} v_i\}\\ \texttt{module}\ m\ \langle x_i\rangle\ \langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \in \langle D_i\rangle\\ \{\langle D_i\rangle \vdash P_k\{[x_i \mapsto v_i]\} \overset{1}{\hookrightarrow} P'_k, \langle D_r\rangle^{r\in R(k)}\}\\ m' \notin \langle D_i\rangle \qquad m' \notin \biguplus_k \langle D_r\rangle^{r\in R(k)}\end{array}}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} m'\langle\rangle\langle l_j\rangle, \langle \texttt{module}\ m'\ \langle\rangle\ \langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P'_k\rangle\rangle \uplus \biguplus_k \langle D_r\rangle^{r\in R(k)}}\ (\text{E}-\text{ModInst})$$

$$\frac{\exists i.e_i \overset{0}{\hookrightarrow} \texttt{error}}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}\ (\text{E}-\text{MErr1}) \qquad \frac{\texttt{module}\ m\ \langle x_i\rangle\ \langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \notin \langle D_i\rangle}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}\ (\text{E}-\text{MErr2})$$

$$\frac{\texttt{module}\ m\ \langle x_r\rangle\ \langle d_h\ t_h\ s_h\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \in \langle D_i\rangle \quad |\langle e_i\rangle| \neq |\langle x_r\rangle|}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}\ \text{(E–MErr3)}$$

$$\frac{\texttt{module}\ m\ \langle x_i\rangle\ \langle d_h\ t_h\ s_h\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \in \langle D_i\rangle \quad |\langle l_j\rangle| \neq |\langle d_h\ t_h\ s_h\rangle|}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}\ \text{(E–MErr4)}$$

$$\frac{\{e_i \overset{0}{\hookrightarrow} v_i\} \quad \texttt{module}\ m\ \langle x_i\rangle\ \langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \in \langle D_i\rangle \quad \exists k.\langle D_i\rangle \vdash P_k\{[x_i \mapsto v_i]\} \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}\ \text{(E}-\text{MErr5)}$$

$$\frac{l \overset{1}{\hookrightarrow} l' \quad e \overset{1}{\hookrightarrow} e'}{\langle D_i\rangle \vdash \texttt{assign}\ l\ e \overset{1}{\hookrightarrow} \texttt{assign}\ l'\ e', \langle\rangle}\ \text{(E}-\text{Assign)}$$

$$\frac{l \overset{1}{\hookrightarrow} \texttt{error}}{\langle D_i\rangle \vdash \texttt{assign}\ l\ e \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{AssignErr1)} \qquad \frac{e \overset{1}{\hookrightarrow} \texttt{error}}{\langle D_i\rangle \vdash \texttt{assign}\ l\ e \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{AssignErr2)}$$

$$\frac{S \overset{1}{\hookrightarrow} S'}{\langle D_i\rangle \vdash \texttt{always}\ S \overset{1}{\hookrightarrow} \texttt{always}\ S', \langle\rangle}\ \text{(E}-\text{Always)} \qquad \frac{S \overset{1}{\hookrightarrow} \texttt{error}}{\langle D_i\rangle \vdash \texttt{always}\ S \overset{1}{\hookrightarrow} \texttt{error}, \langle\rangle}\ \text{(E}-\text{AlwaysErr)}$$

$\boxed{l \overset{1}{\hookrightarrow} l_\perp}$ See $e \overset{1}{\hookrightarrow} e_\perp$

$\boxed{S \overset{1}{\hookrightarrow} S_\perp}$

$$\frac{\{E_i \overset{1}{\hookrightarrow} E_i'\} \quad S \overset{1}{\hookrightarrow} S'}{@\ \langle E_i\rangle\ S \overset{1}{\hookrightarrow} @\ \langle E_i'\rangle\ S'}\ \text{(E}-\text{Trigger)} \qquad \frac{\exists i.E_i \overset{1}{\hookrightarrow} \texttt{error}}{@\ \langle E_i\rangle\ S \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{TrigErr1)} \qquad \frac{S \overset{1}{\hookrightarrow} \texttt{error}}{@\ \langle E_i\rangle\ S \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{TrigErr2)}$$

$$\frac{l \overset{1}{\hookrightarrow} l' \quad e \overset{1}{\hookrightarrow} e'}{l = e \overset{1}{\hookrightarrow} l' = e'}\ \text{(E}-\text{SAssign)} \qquad \frac{l \overset{1}{\hookrightarrow} \texttt{error}}{l = e \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{SAssignErr1)} \qquad \frac{e \overset{1}{\hookrightarrow} \texttt{error}}{l = e \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{SAssignErr2)}$$

$$\frac{e \overset{1}{\hookrightarrow} e' \qquad S_1 \overset{1}{\hookrightarrow} S_1' \qquad S_2 \overset{1}{\hookrightarrow} S_2'}{\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \overset{1}{\hookrightarrow} \texttt{if}\ e'\ \texttt{then}\ S_1'\ \texttt{else}\ S_2'}\ \text{(E}-\text{If)} \qquad \frac{e \overset{1}{\hookrightarrow} \texttt{error}}{\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{IfErr1)}$$

$$\frac{\exists i \in \{1,2\}.S_i \overset{1}{\hookrightarrow} \texttt{error}}{\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{IfErr2)}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v_1 \quad e_2[y \mapsto v_1] \overset{0}{\hookrightarrow} v_2 \qquad v_2 \neq 0^{32} \quad S[y \mapsto v_1] \overset{1}{\hookrightarrow} S' \quad \texttt{for}(y = e_3[y \mapsto v_1];\ e_2;\ y = e_3)\ S \overset{1}{\hookrightarrow} \langle S_i\rangle}{\texttt{for}(y = e_1;\ e_2;\ y = e_3)\ S \overset{1}{\hookrightarrow} S' :: \langle S_i\rangle}\ \text{(E}-\text{ForTrue)}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v \quad e_2[y \mapsto v] \overset{0}{\hookrightarrow} 0^{32}}{\texttt{for}(y = e_1;\ e_2;\ y = e_3)\ S \overset{1}{\hookrightarrow} \langle\rangle}\ \text{(E}-\text{ForFalse)}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \texttt{error}}{\texttt{for}(y = e_1;\ e_2;\ y = e_3)\ S \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{ForErr1)} \qquad \frac{e_1 \overset{0}{\hookrightarrow} v_1 \quad e_2[y \mapsto v_1] \overset{0}{\hookrightarrow} \texttt{error}}{\texttt{for}(y = e_1;\ e_2;\ y = e_3)\ S \overset{1}{\hookrightarrow} \texttt{error}}\ \text{(E}-\text{ForErr2)}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v_1 \quad S[y \mapsto v_1] \overset{1}{\hookrightarrow} \texttt{error}}{\texttt{for}(y = e_1;\ e_2;\ y = e_3)\ S \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{ForErr3}) \qquad \frac{e_1 \overset{0}{\hookrightarrow} v_1 \quad \texttt{for}(y = e_3[y \mapsto v_1];\ e_2;\ y = e_3)\ S \overset{1}{\hookrightarrow} \texttt{error}}{\texttt{for}(y = e_1;\ e_2;\ y = e_3)\ S \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{ForErr4})$$

$$\frac{\{S_i \overset{1}{\hookrightarrow} S_i'\}}{\langle S_i \rangle \overset{1}{\hookrightarrow} \langle S_i' \rangle}\ (\text{E}-\text{SSeq}) \qquad \frac{\exists i. S_i \overset{1}{\hookrightarrow} \texttt{error}}{\langle S_i \rangle \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{SSeqErr})$$

$\boxed{E \overset{1}{\hookrightarrow} E_\perp}$

$$\frac{l \overset{1}{\hookrightarrow} l'}{g\ l \overset{1}{\hookrightarrow} g\ l'}\ (\text{E}-\text{Event}) \qquad \frac{l \overset{1}{\hookrightarrow} \texttt{error}}{g\ l \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{EventErr})$$

$\boxed{e \overset{1}{\hookrightarrow} e_\perp}$

$$\frac{}{s \overset{1}{\hookrightarrow} s}\ (\text{E}-\text{Id1}) \qquad \frac{e \overset{0}{\hookrightarrow} v}{s[e] \overset{1}{\hookrightarrow} s[v]}\ (\text{E}-\text{Index1}) \qquad \frac{e \overset{0}{\hookrightarrow} \texttt{error}}{s[e] \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{Index1Err})$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v_1 \quad e_2 \overset{0}{\hookrightarrow} v_2}{s[e_1 : e_2] \overset{1}{\hookrightarrow} s[v_1 : v_2]}\ (\text{E}-\text{Range1}) \qquad \frac{\exists i. e_i \overset{0}{\hookrightarrow} \texttt{error}}{s[e_1 : e_2] \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{Range1Err}) \qquad \frac{}{x \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{Param1})$$

$$\frac{}{v \overset{1}{\hookrightarrow} v}\ (\text{E}-\text{Int1}) \qquad \frac{\{e_i \overset{1}{\hookrightarrow} e_i'\}}{f\langle e_i \rangle \overset{1}{\hookrightarrow} f\langle e_i' \rangle}\ (\text{E}-\text{Op1}) \qquad \frac{\exists i. e_i \overset{1}{\hookrightarrow} \texttt{error}}{f\langle e_i \rangle \overset{1}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{Op1Err})$$

$\boxed{e \overset{0}{\hookrightarrow} e_\perp}$

$$\frac{}{s \overset{0}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{Id0}) \qquad \frac{}{s[e] \overset{0}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{Index0}) \qquad \frac{}{s[e_1 : e_2] \overset{0}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{Range0})$$

$$\frac{}{x \overset{0}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{Param0}) \qquad \frac{}{v \overset{0}{\hookrightarrow} v}\ (\text{E}-\text{Int0}) \qquad \frac{\{e_i \overset{0}{\hookrightarrow} v_i\}}{f\langle e_i \rangle \overset{0}{\hookrightarrow} [\![f]\!]\langle v_i \rangle}\ (\text{E}-\text{Op0}) \qquad \frac{\exists i. e_i \overset{0}{\hookrightarrow} \texttt{error}}{f\langle e_i \rangle \overset{0}{\hookrightarrow} \texttt{error}}\ (\text{E}-\text{Op0Err})$$

5 Technical Results

To show that the type system achieves the goals described in the introduction, we establish three theorems (Theorems 5.1, 5.2, and 5.3).

5.1 Preprocessing Produces Well-Typed Programs

Lemmas 5.1, 5.2, 5.3, 5.4, and 5.5 establish that substition preserves typing for e, l, E, S, and P terms respectively. Lemmas 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, and 5.12 establish that preprocessing preserves typability for e (2 lemmas), l, S, P, and b terms. Theorem 5.1 establishes that preprocessing preserves typability for programs.

Lemma 5.1. *If* $\Gamma, x : d_1\ t_1 \vdash^{n_2} e : d_2\ t_2$ *and* $\Gamma \vdash^{n_1} v : d_1\ t_1$ *then* $\Gamma \vdash^{n_2} e[x \mapsto v] : d_2\ t_2$

Lemma 5.2. *If* $\Gamma, x : d_1\, t_1 \vdash^{n_2} l : d_2\, t_2$ *and* $\Gamma \vdash^{n_1} v : d_1\, t_1$ *then* $\Gamma \vdash^{n_2} l[x \mapsto v] : d_2\, t_2$

Lemma 5.3. *If* $\Gamma, x : d\, t \vdash E$ *and* $\Gamma \vdash^{n} v : d\, t$ *then* $\Gamma \vdash E[x \mapsto v]$

Lemma 5.4. *If* $\Gamma, x : d\, t \vdash S$ *and* $\Gamma \vdash^{n} v : d\, t$ *then* $\Gamma \vdash S[x \mapsto v]$

Lemma 5.5. *If* $\Delta; \Gamma, x : d\, t \vdash P$ *and* $\Gamma \vdash^{n} v : d\, t$ *then* $\Gamma \vdash P[x \mapsto v]$

Lemma 5.6. *If* $\Gamma^+ \vdash^0 e : d\, t$ *and* $e \overset{0}{\hookrightarrow} e'_\perp$ *then* $\Gamma^+ \vdash^0 e' : d\, t$

Lemma 5.7. *If* $\Gamma^+ \vdash^1 e : d\, t$ *and* $e \overset{1}{\hookrightarrow} e'_\perp$ *then* $\Gamma^+ \vdash^1 e' : d\, t$

Lemma 5.8. *If* $\Gamma^+ \vdash^1 l : d\, t$ *and* $l \overset{1}{\hookrightarrow} l'_\perp$ *then* $\Gamma^+ \vdash^1 l' : d\, t$

Lemma 5.9. *If* $\Gamma^+ \vdash E$ *and* $E \overset{1}{\hookrightarrow} E'_\perp$ *then* $\Gamma^+ \vdash E'$

Lemma 5.10. *If* $\Gamma^+ \vdash S$ *and* $S \overset{1}{\hookrightarrow} S'_\perp$ *then* $\Gamma^+ \vdash S'$

Lemma 5.11. *If* $\Delta; \Gamma^+ \vdash P$ *and* $\langle D_i \rangle \vdash P \overset{1}{\hookrightarrow} P'_\perp, \langle D_k \rangle$ *then* $\Delta; \Gamma^+ \vdash P'$ *and* $\Delta \vdash \langle D_k \rangle : \Delta'$

Lemma 5.12. *If* $\Delta \vdash b : M$ *and* $\langle D_i \rangle \vdash b \overset{1}{\hookrightarrow} b'_\perp, \langle D_k \rangle$ *then* $\Delta \vdash b' : M$ *and* $\Delta \vdash \langle D_k \rangle : \Delta'$

Theorem 5.1. *If* $\vdash p$ *and* $p \overset{1}{\hookrightarrow} p'_\perp$ *then* $\vdash p'$

5.2 Preprocessing does not Depend on Wire Values

Because the definition of preprocessing raises an error if a wire value is encountered during preprocessing, the following theorem demonstrates that preprocessing does not depend on wire values.

Theorem 5.2. *If* $\vdash p$ *and* $p \overset{1}{\hookrightarrow} p'_\perp$ *then* $p' \neq \texttt{error}$

5.3 Preprocessing Produces Fully Expanded Terms

To show that preprocessing produces fully expanded terms, we need to define this set of terms formally. The set expanded terms is identical to the set of terms before expansion except that module bodies and instantiations can not have parameters and that `for`-loop statements are not allowed.

$$\begin{array}{lcl} \textit{Expanded Term} \quad \hat{X} & = & \{u \mid u \in X_\perp \quad \wedge Y \in \textit{subterms}(u) \Rightarrow \\ & & ((Y = \langle x_i \rangle^{i \in I}\ \langle d_j\, t_j\, s_j \rangle^{j \in J}\ \texttt{is}\ \langle t_k\, y_k \rangle^{k \in K}\ \langle P_r \rangle^{r \in R} \Rightarrow I = \emptyset) \\ & & \wedge (Y = m\ \langle e_i \rangle^{i \in I}\ \langle l_j \rangle^{j \in J} \ \Rightarrow I = \emptyset) \\ & & \wedge (Y \neq \texttt{for}(y = e; e; y = e) S))\} \end{array}$$

The above definition formalizes this concept by stating formally that for a term to be fully expanded it has to satisfy the following three conditions:

- If it contains a module declaration, then it must be parameter free.

- If it contains a module instantiation, then it must not pass any parameters to it.
- It cannot contain a `for`-loop.

Lemmas 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, and 5.19 establish that the result of preprocessing e (2 lemmas), l, E, S, P, and b terms respectively produces a fully expanded term of the same category. Theorem 5.3 establishes this property for programs.

Lemma 5.13. *If* $e \overset{0}{\hookrightarrow} e'$ *then* $e' \in \hat{e}$

Lemma 5.14. *If* $e \overset{1}{\hookrightarrow} e'$ *then* $e' \in \hat{e}$

Lemma 5.15. *If* $l \overset{1}{\hookrightarrow} l'$ *then* $l' \in \hat{l}$

Lemma 5.16. *If* $E \overset{1}{\hookrightarrow} E'$ *then* $E' \in \hat{E}$

Lemma 5.17. *If* $S \overset{1}{\hookrightarrow} S'$ *then* $S' \in \hat{S}$

Lemma 5.18. *If* $\langle D_i \rangle \vdash P \overset{1}{\hookrightarrow} P', \langle D_k \rangle$ *then* $P' \in \hat{P}$ *and* $\{D_k \in \hat{D}\}$

Lemma 5.19. *If* $\langle D_i \rangle \vdash b \overset{1}{\hookrightarrow} b', \langle D_k \rangle$ *then* $b' \in \hat{b}$ *and* $\{D_k \in \hat{D}\}$

Theorem 5.3. *If* $p \overset{1}{\hookrightarrow} p'$ *then* $p' \in \hat{p}$

6 Conclusions and Future Work

This paper presented a static analysis that can ensure that the use of iteration constructs and module parameters do not interfere with the synthesizability of a design. In addition to providing a clear specification of when the use of such constructs is safe, the paper also defines an operational semantics that specifies how a synthesis tool must handle such constructs.

The framework presented here ensures that the result of expanding any design is well-typed. We expect that this framework will play a key role in providing other guarantees about the results of synthesis, including matching bus sizes, area, timing, and power.

Acknowledgments: We would like to thank Yousra Alkabani for many valuable discussions about Verilog.

References

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[2] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[3] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, 1988.

[4] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000, USA. Available online from:
ftp://cse.ogi.edu/pub/tech-reports/README.html.

[5] Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. available from [4].

[6] Walid Taha and Patricia Johann. Staged notational definitions. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.