Synthesizable High Level Hardware Descriptions

Jennifer GILLENWATER, Gregory MALECHA, Cherif SALAMA, Angela Yun ZHU, and Walid TAHA

Rice University

Jim GRUNDY and John O'LEARY

Intel Strategic CAD Labs

taha@rice.edu

Received 15 November 2009

Abstract Modern hardware description languages support code generation constructs like `generate/endgenerate` in Verilog. These constructs are used to describe regular or parameterized hardware designs and, when used effectively, can make hardware descriptions shorter, more understandable, and more reusable. In practice, however, designers avoid these abstractions because it is difficult to understand and predict the properties of the generated code. Is the generated code even type safe? Is it synthesizable? What physical resources (e.g. combinatorial gates and flip-flops) does it require? It is often impossible to answer these questions without first generating the fully-expanded code. In the Verilog and VHDL communities, this generation process is referred to as *elaboration*.

This paper proposes a disciplined approach to elaboration in Verilog.[*1] By viewing Verilog as a statically typed two-level language, we are able

[*1] An earlier version was presented at PEPM'08. This manuscript includes performance evaluation, complete proofs, and various improvements. This work was supported by the National Science Foundation (NSF) SoD awards 0439017, 0720857, 0747431 and the Semiconductor Research Consortium (SRC) Task ID: 1403.001 (Intel custom project).

to reflect the distinction between values that are known at elaboration time and values that are part of the circuit computation. This distinction is crucial for determining whether generative constructs, such as iteration and module parameters, are used in a synthesizable manner. This allows us to develop a static type system that guarantees synthesizability. The type system achieves safety by performing additional checking on generative constructs and array indices. To illustrate this approach, we develop a core calculus for Verilog that we call Featherweight Verilog (FV) and an associated static type system. We formally define a preprocessing step analogous to the elaboration phase of Verilog, and the kinds of errors that can occur during this phase. Finally, we show that a well-typed design cannot cause preprocessing errors, and that the result of its elaboration is always a synthesizable circuit.

Keywords Code Generation, Hardware Description Languages, Statically Typed Two-Level Languages, Synthesizability, Verilog Elaboration.

§1 Introduction

The Verilog language has three kinds of constructs. Constructs of the first kind are used to describe how the components of a circuit are connected. These are usually referred to as *structural* constructs. Constructs of the second kind allow the description of circuit functionality at an algorithmic level. These are most often used for simulation purposes, and are usually referred to as *behavioral* constructs. The third kind of construct describes the generation of more Verilog code through a processes called *elaboration*. Constructs of this kind are known as *generative* constructs, and they include parameterized modules, conditionals and iteration. Generative constructs provide a high level abstraction mechanism by allowing for compact and reusable descriptions of circuit families.

For example, a *family* of adders such as the one presented in Figure 1 can be described as follows:

```
module adder(s,cout,a,b,cin);
   parameter N=4;
   input [N-1:0] a,b;      input cin;
   output [N-1:0] s;       output cout;
   wire [N:0] c;           genvar i;
   assign c[0] = cin;
   generate
      for(i=0; i<N; i=i+1)
         full_adder fa (s[i],c[i+1],a[i],b[i],c[i]);
   endgenerate
   assign cout = c[N];
endmodule
```

Both module parameterization (`parameter N=4`) and iteration (the `for`-loop) are essential for describing the family of adders.

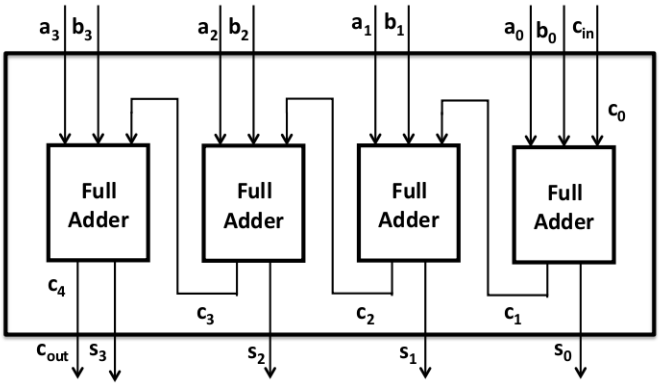

Fig. 1 4-bit ripple adder

The first line starts the description of a new hardware module called `adder`, which has five ports `s`, `cout`, `a`, `b`, and `cin`. Ports are the module's interface with the outside world. The second line declares a parameter `N` whose default value is 4. This declaration makes the module under consideration a generic module that can be instantiated with various values of `N` to create differently-sized circuits. Next, a set of declarations specifies the directions and sizes of each of the ports of this module. For example, `cin` is a one bit input port, while `s` is an `N`-bit output port whose wires are indexed from `N-1` to `0`. The statement `wire [N:0] c` is used to declare an internal array of wires that cannot be seen from the outside of the module. The last declaration `genvar i` is used to declare the loop index.

The circuit's structure is defined by three parallel statements describing its components and their interconnections. The first statement connects the input `cin` to the least significant bit of `c`. Assuming that a `full_adder` module has been defined elsewhere, the generate block instantiates `N` interconnected `full_adder`s using a `for`-loop. The last parallel statement connects the most significant bit of `c` to the `cout` port.

The most important property of a description is whether or not it is physically realizable as a circuit. This property is often referred to as synthesizability. Intuitively, a synthesizable description is a description that has a direct correspondence to a circuit. Unfortunately, the line between synthesizable and non-synthesizable descriptions is unclear and *ad hoc*. The Verilog Register

Transfer Language (RTL) synthesis standard (IEEE 1364.1 [8]) does not formally draw this line. Instead, it gives some synthesizability guidelines, illustrated by a series of good and bad examples. Although this is useful to understand what kind of descriptions should be synthesizable, it is not sufficient.

The first Verilog standard (IEEE 1364-1995 [4]) supported iterations and conditionals only as behavioral statements meant for simulation. Whether these constructs were synthesizable or not was implementation dependent. The Verilog-2001 standard [5], and subsequent Verilog-2005 [7] and System Verilog [6] standards, extend Verilog-95 with the `generate/endgenerate` construct, which allows conditionals and single variable iteration statements to appear in parallel statements. When enclosed in `generate` blocks, these statements are elaborated into ordinary parallel statements prior to simulation or synthesis.

Because current Verilog compilers analyze `generate` blocks only after they are elaborated, errors remain undetected until synthesis, when they are more difficult to diagnose. In addition, only concrete instances of a family of designs — like the default four-wide instance of the ripple adder above — are checked for errors. A similar problem is familiar in programming languages that introduce a stage of code generation before execution. Macros in C and templates in C++ are examples where the characteristics of a program cannot be understood without "expanding away" the macros or templates. For this reason sophisticated use of such features is wisely curtailed by developers despite the obvious power of the technique. So it is with Verilog: In both educational and industrial settings it is common to see the loop in the previous generic design manually unrolled into the following concrete instance:

```
full_adder fa_0 (s[0],c[1],a[0],b[0],c[0]);
full_adder fa_1 (s[1],c[2],a[1],b[1],c[1]);
full_adder fa_2 (s[2],c[3],a[2],b[2],c[2]);
full_adder fa_3 (s[3],c[4],a[3],b[3],c[3]);
```

Alternatively, designers commonly use scripting languages, such as Perl, to generate Verilog code for specific instances of module families. Both unrolling and scripting are undesirable. Unrolling is tedious, error-prone and violates many software engineering principles. Scripting employs a language that manipulates hardware descriptions as strings rendering static analysis of any kind impractical. The generated code is not even guaranteed to be syntactically correct!

However, more disciplined approaches exist: Two-level [3,12] and multi-level languages [15,17] have been studied as a way to understand software code

generation. They provide a formal infrastructure that allows characteristics of programs to be checked without requiring expansion. For example, Kiselyov, Swadi and Taha [9] have shown how to generate highly optimized, type correct Fast Fourier Transform kernel routines from compact algorithmic descriptions written using multi-level languages. Similarly, Taha, Ellner and Xi [16] have shown how to generate heap bounded implementations of sorting programs from a compact, parameterized sorting algorithm written using a two-level language. In both cases all static analysis is performed prior to code generation.

Bluespec SystemVerilog (BSV) [2] uses Term Rewriting Systems (TRS) [18] to provide powerful generate-like features with static checks. However, the relationship between BSV and TRS is not formally explained. A formal semantics is a prerequisite for having static guarantees about synthesizability.

Our thesis is that the techniques developed for statically typed two-level languages are particularly pertinent to hardware description languages.

1.1 Contributions

This paper shows that, by treating Verilog as a statically typed two-level language, we can statically check the synthesizability of a description with high level abstractions (generative constructs) without having to elaborate it. The ability to statically check Verilog descriptions:

- Provides a proof of concept that we can check properties of the circuit generated from elaboration without actually performing elaboration and paves the way for more aggressive static checking.
- Suggests that designers may be able to use high level abstractions without sacrificing the benefits of static checking. For example, any misuse of abstractions that might impair synthesizability will be detected statically.
- Enables the usage of a single, tightly integrated, language to describe circuits and circuit families.
- Enables the usage of the same type checker to check descriptions before and after elaboration.
- Enables checking the synthesizability of families of circuits once, rather than at each instantiation.

To achieve static checking, we provide a rigorous definition for two notions of synthesizability, namely, *obvious synthesizability* and *general synthesizability* in an implementation independent way (Section 2.5). These concepts allow us to pin down the Verilog constructs that are interesting from the syn-

thesizability point of view and formally treat them.

To address the above goals from a semantic point of view, we define Featherweight Verilog (FV), a calculus for a representative core of structural Verilog (Section 2). A two-level operational semantics for FV captures how various generative constructs should be elaborated and what can go wrong during this process (Section 3). A two-level type system is used to define the conditions needed to guarantee that various constructs do not interfere with synthesis (Section 4).

We establish three properties of FV (Section 5). Theorem 1 states that elaboration preserves typing. Theorem 2 states that it is always safe to perform elaboration of a well-typed design by showing that elaboration never depends on wire values. Theorem 3 states that the result of elaboration is obviously synthesizable. Combined, these results establish that a well-typed design is synthesizable, which implies that we can statically check for synthesizability of a description before elaborating it using a relatively simple type checker. Auxiliary results and a complete formal proof for each of these theorems appear in (Appendix 3).

A prototype implementation in the form of a Verilog Pre-Processor (VPP) was developed. VPP statically checks the synthesizability of a Verilog description (possibly containing high level abstractions) and elaborates it if it is well-typed (Section 6).

§2 Syntax of FV and Synthesizable Subset

Because synthesizability is central to this study, FV formalizes primarily the structural subset of Verilog as opposed to the behavioral subset that is primarily intended for simulation and testing. Moreover, whereas the full Verilog language provides many constructs to make writing hardware descriptions as convenient as possible, a calculus captures the essence and not the totality of the language. FV models signals, primitive gates, conditionals, iterations, and parameterized modules.

2.1 Notational Conventions

The following notational conventions are used in the rest of this paper:

- A sequence is either the empty sequence $\langle\rangle$ or a non-empty sequence $h :: t$ with a head h and a tail sequence t.
- 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.

- 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 $|\, S \,|$ for the length of the sequence S.

2.2 Formal Syntax (BNF)

The abstract syntax for FV makes use of the following meta-variables:

Module	m	$\in$	ModuleNames
Signal	s	$\in$	IdentifierNames
Elaboration Variable	x, y	$\in$	ParameterNames
Operator	f	$\in$	$\mathbb{O}$
Index	h, i, j, k, q, r	$\in$	$\mathbb{N}$
Index Domain	H, I, J, K, Q, R	$\subseteq$	$\mathbb{N}$

where ModuleNames, IndentifierNames, and ParameterNames are countably infinite sets used to draw modules, signals, and parameters names respectively. The set of operator names $\mathbb{O}$ is finite, and $\mathbb{N}$ is the set of natural numbers. The entire grammar for FV is defined as follows:

Circuit Description	p	::=	$\langle D_i \rangle^{i \in I}\ m$
Module Definition	D	::=	`module` $m\ b$
Module Body	b	::=	$\langle x_i \rangle^{i \in I}\ \langle d_j\ s_j \rangle^{j \in J}$ `is` $\langle t_k\ s_k \rangle^{k \in K}\ \langle P_r \rangle^{r \in R}$
Direction	d	$\subseteq$	{`in`, `out`}
Type	$t \in \mathbb{T}$	::=	`wire` | `int`
Parallel Statement	P	::=	$m\ \langle e_i \rangle^{i \in I}\ \langle l_j \rangle^{j \in J}$ | `assign` $l\ e$ | `if` e `then` $\langle P_i \rangle^{i \in I}$ `else` $\langle P_j \rangle^{j \in J}$ | `for` $(y = e; e; y = e) \langle P_i \rangle^{i \in I}$
LHS value	l	::=	$s \mid s[e] \mid s[e : e]$
Expression	e	::=	$l \mid x \mid v \mid f \langle e_i \rangle^{i \in I}$
Value	v	::=	$(0 \,|\, 1)^{32}$

A circuit description p 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 represents the overall input and output of the system. A module definition consists of a name m and a module body b. A module body itself consists of four sequences: (1) module parameter names $\langle x_i \rangle^{i \in I}$, (2) port declarations (carrying direction, and name for each port) $\langle d_j\ s_j \rangle^{j \in J}$, (3) local variable declarations (carrying type, and name for each variable) $\langle t_k\ s_k \rangle^{k \in K}$, and (4) parallel statements $\langle P_r \rangle^{r \in R}$. A port direction indicates whether the

port is input, output, or bidirectional. The type of a local variable can be either `wire`, if the variable represents an internal physical connection, or `int` otherwise. A parallel statement can be a module instantiation, an `assign` statement, a conditional statement, or a `for`-loop. A module instantiation specifies module parameters $\langle e_i \rangle^{i \in I}$ as well as port connections $\langle l_j \rangle^{j \in J}$. An `assign` statement consists of a left hand side (LHS) value l and an expression e. An LHS value is either a variable s, an array lookup $s[e]$, or an array range $s[e : e]$. An expression is either an LHS value l, a parameter name x, a 32-bit integer v, or an operator application $f\langle e_i \rangle^{i \in I}$.

For notational convenience, we define a general term X that is used to range over all syntactical constructs of FV as follows:

$$\textit{Term} \qquad X \quad ::= \quad p \mid D \mid b \mid P \mid l \mid e$$

2.3 Relation of Calculus to Verilog

The calculus resembles and simplifies the concrete syntax for Verilog. FV deviates from Verilog as follows:

- All sequences are represented uniformly as `<a,b,...,z>`.
- The start of the module body is marked with the terminal "`is`".
- Local variable declarations are aggregated immediately after the `is` terminal.
- Module parameters are listed as arguments rather than being declared locally.
- Module parameters do not have default values and so values must always be provided explicitly with each use.
- The direction of a port is declared as part of the formal module argument (as allowed starting from Verilog-2001), rather than in the body of the module definition (as in Verilog-95).
- Verilog keywords, such as `input` and `output`, are replaced by shorter names, such as `in` and `out`.
- Variable direction is represented using a set instead of a keyword. The sets {`in`}, {`out`}, or {`in`, `out`} replace the keywords `in`, `out`, and `inout` respectively. The last set is also used for non-directional variables such as internal signals.
- The `if`-statement always has an `else` clause, but that clause can contain an empty sequence of statements.

- Wire sizes are dropped from terms because Verilog uses automatic coercions to pad arrays of wires of different size to match them.
- All `for`-loop variables are declared implicitly and are local to the loop.
- Primitive gates are not explicitly modeled, but they can be expressed using logical operators.
- Integers are represented in binary.

2.4 Example

The ripple adder module presented in the introduction can be written using the calculus as follows:

```
module adder <N> <out s, out cout, in a, in b, in cin> is
   <wire c>
   <assign c[0] cin,
     for(i=0; i<N; i=i+1)
         < full_adder <> <s[i],c[i+1],a[i],b[i],c[i]> >,
     assign cout c[N]>
```

As explained earlier, there are no default values for parameters and no wire sizes in the calculus. This syntax is more concise and more suitable for formal treatment.

2.5 Synthesizable Subsets

To proceed, we must specify what is synthesizable in FV and what is not. To do so we introduce and define two general concepts:

- **obvious synthesizability** means that a description uniquely determines a directed graph where nodes are either primitive gates or obviously synthesizable modules and edges are wires connecting them.
- **general synthesizability** (or **synthesizability** for short) means that a description is either obviously synthesizable or will become obviously synthesizable after elaboration.

Note that uniqueness of the graph does not imply a unique hardware implementation (because there are different libraries and even different ways to implement a circuit using a given library). Instead, it means that there is a systematic and deterministic way to convert the description to a graph representing the circuit. These definitions are applicable to any hardware description language in general and are implementation independent. Thus descriptions that are obviously synthesizable according to this definition should be synthesizable by all sensible synthesis tools supporting the language in which the description is written.

For Verilog, when structural descriptions are free from high level abstractions, they are obviously synthesizable. The same applies to FV as well since it is a subset of structural Verilog: Well-formed FV descriptions free from abstraction (parameterized modules, conditionals, and `for`-loops) are obviously synthesizable. An FV description is well-formed if it is syntactically correct and satisfies a few conditions that are captured by our type system as defined in section 4.

§3 Operational Semantics for Preprocessing

In the terminology of two-level languages, *preprocessing*[*2] is the level 0 computation, and the result after preprocessing is the level 1 computation. The latter being the computation preformed by the generated circuit. Preprocessing only eliminates generative constructs. As a result, generative constructs are level 0 computations. The rest of the constructs are level 1 computations, and remain unchanged during preprocessing. Preprocessing a term at level 0 involves evaluating some of its subexpressions. There are relatively few places where evaluation is required: 1) expressions passed as module parameters, 2) conditional expressions in `if` statements, 3) expressions that relate to the bounds on `for`-loops, and 4) array indices. Expressions appearing on the right hand side of assignment expressions are not evaluated during preprocessing.

We use a big-step operational semantics indexed by the level of the computation to formally specify the preprocessing phase. The specification dictates how preprocessing should be performed, what the form of the preprocessed circuit descriptions should be, and what errors can occur during preprocessing.

To model the possibility of errors during preprocessing, we define the following auxiliary notion:

$$\textit{Possible Term} \qquad X_\perp \quad ::= \quad X \mid \texttt{err}$$

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

We also need to define a notion of preprocessed terms which defines the preprocessing output language. The set of preprocessed terms is defined as follows:

[*2] Despite the fact that the term *preprocessing* is often used to refer to rather *ad hoc* tools like the C preprocessor (cpp), we prefer to use it for our highly disciplined approach because it implicitly conveys the light weight, unobtrusive quality that VPP is designed to achieve.

$$
\begin{aligned}
&\textit{Preprocessed Term } \hat{X} \;=\; \\
&\{u \mid u \in X_\perp \wedge \forall Y.(Y \in \textit{subterms}(u) \Rightarrow \\
&\quad ((Y = \langle x_i \rangle^{i \in I}\ \langle d_j\ s_j \rangle^{j \in J}\ \texttt{is}\ \langle t_k\ y_k \rangle^{k \in K}\ \langle P_r \rangle^{r \in R} \Rightarrow I = \emptyset) \\
&\quad \wedge (Y = m\ \langle e_i \rangle^{i \in I}\ \langle l_j \rangle^{j \in J}\ \Rightarrow I = \emptyset) \\
&\quad \wedge (Y = s[e_1 : e_2] \Rightarrow e_1 \in v \wedge e_2 \in v) \\
&\quad \wedge (Y = s[e] \Rightarrow e \in v) \\
&\quad \wedge (Y \neq \texttt{if}\ e\ \texttt{then}\langle P_i \rangle^{i \in I}\texttt{else}\langle P_j \rangle^{j \in J}) \\
&\quad \wedge (Y \neq \texttt{for}(y = e; e; y = e)\langle P_i \rangle^{i \in I})))\}
\end{aligned}
$$

This definition says that a preprocessed term is one where all subterms have the following properties:

- A module definition has no parameters.
- Module instantiations do not pass any parameters.
- An array index is a 32-bit value.
- There are no `for`-loops.
- There are no `if`-statements.

A notion of substitution is also required. $X[x \mapsto v]$ denotes the substitution of x by v in X. For FV, the substitution definition is straightforward and can be found with a brief discussion in Appendix 1.

Preprocessing is defined by the derivability of judgments of the general form $\langle D_i \rangle \vdash X \overset{n}{\hookrightarrow} X_\perp, \langle D_j \rangle$. Intuitively, preprocessing takes a sequence of module declarations $\langle D_i \rangle$ and a term X and produces a new sequence of specialized modules $\langle D_j \rangle$ and a possible term $X_\perp$. The input modules $\langle D_i \rangle$ can be dropped when we process an entity that does not require knowledge about the modules available in the context, and the generated modules $\langle D_j \rangle$ can be omitted when we process an entity that cannot instantiate new modules. The value of n can either be 1, to indicate preprocessing a term at level 1, or 0, to indicate preprocessing a term at level 0. In FV, preprocessing a term at level 0 is only used with expressions and is equivalent to evaluating them. Expression evaluation always results in an integral value or an error.

Successful preprocessing is defined in Figure 2. These rules formalize the following. Preprocessing a circuit description (E-Prog) starts by preprocessing the main module and other modules are instantiated as needed. Preprocessing the body of the main module (E-Body) involves preprocessing each of its

statements. The curly braces surrounding the preprocessing judgement indicate that this is a set of judgements of size $|K|$, one for each parallel statement P_k. Preprocessing a statement P_k can generate several statements $\langle P_r \rangle^{r \in R(k)}$ and can involve instantiating several modules $\langle D_h \rangle^{h \in H(k)}$. The set $R(k)$ is the set of indices of the resulting statements. This set is a function of k since it will be different for each of the parallel statements preprocessed. Similarly $H(k)$ is the set of indices of the resulting modules. All the obtained statements and modules are aggregated (in order) and returned.

Preprocessing a module instantiation (E-Mod) generates a new module representing a unique instance of the module definition. The chosen name m' for the generated module must be globally unique. The rules for preprocessing the assignment statement (E-Assign) and conditionals (E-IfTrue and E-IfFalse) are straightforward. Preprocessing a `for`-loop (E-ForTrue and E-ForFalse) amounts to executing the `for`-loop, except that the result of execution is a sequence of statements rather than a modification of the global state.

Expressions that are at level 1 (E-Id, E-Idx, E-Rg, E-Int1, and E-Op1) are recursively preprocessed without evaluating them, and ones at level 0 (E-Int0 and E-Op0) are evaluated. For expressions at level 0, the only active rule in evaluation pertains to operator applications denoted by $[\![f]\!]\langle v_i \rangle$ (E-Op0).

Because we want to prove that our type system ensures that no errors can occur during preprocessing, we need to define error cases and error propagation explicitly. Preprocessing errors are defined in Appendix 2.

An example of a preprocessing error occurs when preprocessing the following description:

```
module badinv (q,a,n);
   input [3:0] n;     input [15:0] a;
   output [15:0] q;   genvar i;

   generate
      for(i=0; i<=n; i=i+1)
         assign q[i] = ~a[i];
   endgenerate
endmodule
```

The previous example describes a module `badinv` with one output port `q` and two input ports `a` and `n`. This circuit is supposed to inverts each of the input bits of `a`, connecting the result to the corresponding output bit in `q`. The `~` operator is Verilog's inversion operator. Clearly this circuit cannot be realized because the number of inverters composing it is determined by the value on the

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

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

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

$$\frac{\{\langle D_i\rangle \vdash P_k \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(k)}, \langle D_h\rangle^{h\in H(k)}\}}{\begin{array}{l}\langle D_i\rangle \vdash \langle\rangle\langle d_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\langle P_k\rangle \\ \quad \overset{1}{\hookrightarrow} \langle\rangle\langle d_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \biguplus_k \langle P_r\rangle^{r\in R(k)}, \biguplus_k \langle D_h\rangle^{h\in H(k)}\end{array}}\ \text{(E-Body)}$$

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

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

$$\frac{\begin{array}{l}\{e_i \overset{0}{\hookrightarrow} v_i\} \qquad \{l_j \overset{1}{\hookrightarrow} l'_j\} \qquad m' \textit{ is globally unique} \\ \texttt{module}\ m\ \langle x_i\rangle\ \langle d_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \in \langle D_i\rangle \\ \{\langle D_i\rangle \vdash P_k\{[x_i \mapsto v_i]\} \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(k)}, \langle D_h\rangle^{h\in H(k)}\}\end{array}}{\begin{array}{l}\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \\ \quad \overset{1}{\hookrightarrow} \langle m'\langle\rangle\langle l'_j\rangle\rangle, \biguplus_k \langle D_h\rangle^{h\in H(k)} \uplus \langle \texttt{module}\ m'\ \langle\rangle\ \langle d_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \biguplus_k \langle P_r\rangle^{r\in R(k)}\rangle\end{array}}\ \text{(E-Mod)}$$

$$\frac{e \overset{0}{\hookrightarrow} v \qquad v \neq 0^{32} \qquad \{\langle D_i\rangle \vdash P_k \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(k)}, \langle D_h\rangle^{h\in H(k)}\}}{\langle D_i\rangle \vdash \texttt{if}\ e\ \texttt{then}\ \langle P_k\rangle\ \texttt{else}\ \langle P_j\rangle \overset{1}{\hookrightarrow} \biguplus_k \langle P_r\rangle^{r\in R(k)}, \biguplus_k \langle D_h\rangle^{h\in H(k)}}\ \text{(E-IfTrue)}$$

$$\frac{e \overset{0}{\hookrightarrow} 0^{32} \qquad \{\langle D_i\rangle \vdash P_j \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(j)}, \langle D_h\rangle^{h\in H(j)}\}}{\langle D_i\rangle \vdash \texttt{if}\ e\ \texttt{then}\ \langle P_k\rangle\ \texttt{else}\ \langle P_j\rangle \overset{1}{\hookrightarrow} \biguplus_j \langle P_r\rangle^{r\in R(j)}, \biguplus_j \langle D_h\rangle^{h\in H(j)}}\ \text{(E-IfFalse)}$$

$$\frac{\begin{array}{l}e_1 \overset{0}{\hookrightarrow} v_1 \qquad e_2[y \mapsto v_1] \overset{0}{\hookrightarrow} v_2 \qquad v_2 \neq 0^{32} \\ \{\langle D_i\rangle \vdash P_k[y \mapsto v_1] \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(k)}, \langle D_h\rangle^{h\in H(k)}\} \\ \langle D_i\rangle \vdash \texttt{for}(y = e_3[y \mapsto v_1];\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle P_j\rangle, \langle D_q\rangle\end{array}}{\langle D_i\rangle \vdash \texttt{for}(y = e_1;\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \biguplus_k \langle P_r\rangle^{r\in R(k)} \uplus \langle P_j\rangle, \biguplus_k \langle D_h\rangle^{h\in H(k)} \uplus \langle D_q\rangle}\ \text{(E-ForTrue)}$$

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

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

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

$$\frac{}{s \overset{1}{\hookrightarrow} s}\ \text{(E-Id)} \qquad \frac{e \overset{0}{\hookrightarrow} v}{s[e] \overset{1}{\hookrightarrow} s[v]}\ \text{(E-Idx)} \qquad \frac{e_1 \overset{0}{\hookrightarrow} v_1 \qquad e_2 \overset{0}{\hookrightarrow} v_2}{s[e_1 : e_2] \overset{1}{\hookrightarrow} s[v_1 : v_2]}\ \text{(E-Rg)}$$

$$\frac{}{v \overset{1}{\hookrightarrow} v}\ \text{(E-Int1)} \qquad \frac{\{e_i \overset{1}{\hookrightarrow} e'_i\}}{f\langle e_i\rangle \overset{1}{\hookrightarrow} f\langle e'_i\rangle}\ \text{(E-Op1)}$$

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

$$\frac{}{v \overset{0}{\hookrightarrow} v}\ \text{(E-Int0)} \qquad \frac{\{e_i \overset{0}{\hookrightarrow} v_i\}}{f\langle e_i\rangle \overset{0}{\hookrightarrow} [\![f]\!]\langle v_i\rangle}\ \text{(E-Op0)}$$

Fig. 2 Operational Semantics

input wire `n`. In the next section, we define the type system that allows us to statically detect such violations.

§4 Type System

This section presents a type system for FV. It specifies how to type check descriptions making use of abstraction mechanisms such as iterations, conditionals, and module parameters. In the next section, we will show that it guarantees synthesizability.

By convention, 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 that has already been preprocessed. Expressions, however, may be computations that either are performed during preprocessing or remain intact to become part of the preprocessed description. For this reason, the judgment for expressions will be annotated with a level $n \in \{0, 1\}$ to indicate whether we are checking the expression for validity at level 0 or at level 1. This annotation will appear in the judgment as a superscript on the turnstyle, i.e. $\vdash^n$.

4.1 Typing Environments

To define the type system we need the following auxiliary notions:

Module Type	M	::=	$k\ \langle d_i \rangle^{i \in I}$
Operator Signatures	Σ	$\in$	$\Pi i.\ \mathbb{O} \times \mathbb{N} \times \mathbb{T}^i \to \mathbb{T}$
Level	n	::=	$0 \mid 1$
Module Environment	Δ	::=	$[] \mid (m : M) :: \Delta$
Variable Environment	Γ	::=	$[] \mid (s : d\ t) :: \Gamma \mid (x : d\ t) :: \Gamma$
Level 1 Variable Env.	Γ^+	::=	$[] \mid (s : d\ t) :: \Gamma^+$

A module type consists of the number of its parameters and a sequence of directions for ports. The signature of an operator of arity i is a function that takes an operator, the level at which the operation is executed, and 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 module names with their corresponding types while a variable environment associates variable names with their corresponding directions and types. Environments do not keep track of levels because they are determined by the variable type: All signals and declared local variables (denoted by s) are level 1 variables while parameters and `for`-loop variables (denoted by x or y) are level 0 variables.

4.2 Typing Rules

Figure 3 defines the rules for the judgment $\vdash p$ which specifies when a circuit description p is well-typed. The typing rules formalize the following requirements. A circuit description p is typable when the declarations it contains produce a valid module environment and the main module has a type that involves no module parameters (T-Prog). This means that all the modules are well-typed and the top module is not parameterized since this module is automatically instantiated (recall that in FV, parameters do not have default values).

$\boxed{\vdash p}$

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

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

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

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

$$\frac{\begin{array}{l}\{d_j \neq \emptyset\} \qquad \{\Delta; \Gamma \vdash P_r\} \\ \Gamma = \langle x_i : \{\texttt{in}\}\ \texttt{int} \rangle \uplus \langle s_j : d_j\ \texttt{wire} \rangle \uplus \langle s_k : \{\texttt{in}, \texttt{out}\}\ t_k \rangle\end{array}}{\Delta \vdash \langle x_i \rangle\ \langle d_j\ s_j \rangle\ \texttt{is}\ \langle t_k\ s_k \rangle\ \langle P_r \rangle : |\langle x_i \rangle|\ \ \langle d_j \rangle}\ \text{(T-Body)}$$

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

$$\frac{\begin{array}{l}\{\Gamma \vdash^0 e_i : \{\texttt{in}\}\ \texttt{int}\} \qquad \Delta(m) = |\,\langle e_i \rangle\,|\,\langle d_j \rangle \\ \{\Gamma \vdash^1 l_j : d'_j\ t_j\} \qquad \{d_j \subseteq d'_j\}\end{array}}{\Delta; \Gamma \vdash m \langle e_i \rangle\ \langle l_j \rangle}\ \text{(T-Mod)}$$

$$\frac{\texttt{out} \in d_1 \qquad \Gamma \vdash^1 l : d_1\ t_1 \qquad \texttt{in} \in d_2 \qquad \Gamma \vdash^1 e : d_2\ t_2}{\Delta; \Gamma \vdash \texttt{assign}\ l\ e}\ \text{(T-Assign)}$$

$$\frac{\Gamma \vdash^0 e : \{\texttt{in}\}\ \texttt{int} \qquad \{\Delta; \Gamma \vdash P_i\} \qquad \{\Delta; \Gamma \vdash P_j\}}{\Delta; \Gamma \vdash \texttt{if}\ e\ \texttt{then} \langle P_i \rangle \texttt{else} \langle P_j \rangle}\ \text{(T-If)}$$

$$\frac{\begin{array}{l}\Gamma, y : \{\texttt{in}\}\ \texttt{int} \vdash^0 e_2, e_3 : \{\texttt{in}\}\ \texttt{int} \\ \Gamma \vdash^0 e_1 : \{\texttt{in}\}\ \texttt{int} \qquad \{\Delta; \Gamma, y : \{\texttt{in}\}\ \texttt{int} \vdash P_i\}\end{array}}{\Delta; \Gamma \vdash \texttt{for}(y = e_1; e_2; y = e_3) \langle P_i \rangle}\ \text{(T-For)}$$

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

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

$$\frac{\Gamma(s) = d\ t}{\Gamma \vdash^1 s\ :\ d\ t}\ \text{(T-Id)} \qquad \frac{\Gamma(s) = d\ t \qquad \Gamma \vdash^0 e : \{\texttt{in}\}\ \texttt{int}}{\Gamma \vdash^1 s[e] : d\ t}\ \text{(T-Idx)}$$

$$\frac{\begin{array}{l}\Gamma(s) = d\ t \\ \Gamma \vdash^0 e_1, e_2 : \{\texttt{in}\}\ \texttt{int}\end{array}}{\Gamma \vdash^1 s[e_1\ :\ e_2] : d\ t}\ \text{(T-Rg)} \qquad \frac{\Gamma(x) = \{\texttt{in}\}\ \texttt{int}}{\Gamma \vdash^n x\ :\ \{\texttt{in}\}\ \texttt{int}}\ \text{(T-Par)}$$

$$\frac{}{\Gamma \vdash^n v : \{\texttt{in}\}\ \texttt{int}}\ \text{(T-Int)} \qquad \frac{\{\Gamma \vdash^n e_i : d\ t_i\}}{\Gamma \vdash^n f \langle e_i \rangle : \{\texttt{in}\}\ \Sigma |\langle e_i \rangle| (f, n, \langle t_i \rangle)}\ \text{(T-Op)}$$

Fig. 3 Type System

T-MEmpty and T-MSeq define a well-typed module sequence. The body of each module in the sequence must be well-typed under the context of the previous modules.

To type the body of a module, we check that each parallel statement is typable in the context of the current module environment (Δ) and a new variable environment (Γ) composed of the formal parameters and the local variables (T-Body). Local variables are treated as `inout` signals, ports are considered to be of type `wire`, and variables are added to Γ without specifying their levels since these are syntactically distinguishable.

The next set of rules is used to define well-typed parallel statements based on their kind. We have four different cases: (1) For a module instantiation, the rule (T-Mod) requires that the instantiated module is found in the current module environment and has a type compatible with the number of passed parameters and the number and directions of passed signals. Note that the expressions passed as module parameters (if any) must be typable as level 0 computations. (2) For an assignment, the rule (T-Assign) requires that both LHS and RHS expressions are typable at level 1 since the assignment is a computation performed by the synthesized circuit, not during elaboration. The rule for `assign` is somewhat peculiar, because it does not require that t_1 and t_2 are the same. The Verilog type system does not enforce that wire sizes match because of the padding semantics for wires of different sizes. (3) For a conditional, the rule (T-If) requires that the conditional expression is typable at level 0 with type `int` and that each of the parallel statements forming the consequent and the alternative is typable at level 1. (4) For a loop, the rule (T-For) requires that the 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 an integer (with direction {`in`}). Finally, each of the parallel statements forming the loop body must be typable at level 1 under that extended environment.

The rules for expressions are the most intricate. They allow LHS values to be typed only at level 1 (T-Id, T-Idx and T-Rg). In the case of T-Idx and T-Rg, the rules additionally require that the indices be typable at level 0 with type `int`. The rules for T-Par, T-Int and, T-Op always use {`in`} as the direction of the expression under consideration since it is "readable". The rule for operators (T-Op) implicitly requires that the operator name and its associated typing are found in Σ.

By specifying which expressions need to be typable at level 0, these typing rules guarantee the static availability of all the information needed to get rid of the abstractions during elaboration. By doing so, the type system guarantees the success of elaboration for all well-typed descriptions as well as the success of its synthesis as will be shown in section 5.

Returning to the `badinv` example, we can see how it will be statically rejected by our type system. According to T-Mod, for this program to be well-typed, `n` should be typable at level 0 with type `{in} int` and, since there are no rules for typing a signal at level 0, our type system successfully detects that this program is not well-typed and therefore cannot guarantee its synthesizability.

4.3 Simplifying Assumptions

The type system leaves out two conditions that are necessary to guarantee synthesizability:

- Termination of `for`-loops.
- Consistency of wire assignments (Each wire must be assigned exactly once).

It is possible to add restrictions on `for`-loops to ensure termination and to use a linear type system to avoid inconsistent assignments. Both issues, however, are orthogonal to the problems addressed by our type system and we expect that they can easily be checked by other techniques. We choose not to include these checks in our type system to avoid the associated complexity. As such, this work is only a first step toward a complete static synthesizability checker.

§5 Technical Results

We establish three theorems whose complete proofs are presented in the paper appendix. We first show that the preprocessing of a well-typed description produces a well-typed description. Formally:

Theorem 5.1 (Type Preservation)
If $\vdash p$ and $p \overset{1}{\hookrightarrow} p'$ then $\vdash p'$

Proof [Sketch] The proof proceeds by induction on the derivation of the second judgment. ■

The most interesting cause of preprocessing errors in our setting is when a preprocessing computation depends on a wire value. This cannot occur for a

well-typed term.

Theorem 5.2 (Type Safety)
If $\vdash p$ and $p \overset{1}{\hookrightarrow} p'$ then $p' \neq \texttt{err}$

Proof This result follows directly from Theorem 5.1 since no typing rules will consider `err` well-typed. ■

Theorem 5.3 establishes the soundness of preprocessing which refers to the property that elaboration produces fully preprocessed descriptions.

Theorem 5.3 (Preprocessing Soundness)
If $p \overset{1}{\hookrightarrow} p'$ then $p' \in \hat{p}$

Proof [Sketch] The proof proceeds by induction on the derivation of the first judgment. ■

As stated in Section 2.5, well-typed FV programs free from abstractions are obviously synthesizable. Combining this observation with the last three theorems means that we can use our type system to check for the synthesizability of a circuit description statically prior to elaboration: If an FV program is well-typed, Theorem 5.2 says its elaboration will not produce an error and Theorems 5.1 and 5.3 say that the result will be well-typed and abstraction-free. The result of elaborating a well-typed FV program is therefore obviously synthesizable.

§6 Experimental Results

This section summarizes the main results from our experience with implementing and using the ideas proposed in this paper.

6.1 Implementation

A prototype implementation of the Verilog Pre-Processor (VPP) is available for download at `http://www.resource-aware.org/twiki/bin/view/RAP/VPP`. VPP includes a type checker based on the typing rules defined in this paper. If the description contains abstractions, VPP's type checker determines whether they are used in a synthesizable manner. If the given description is proved synthesizable, then it is elaborated into an equivalent, obviously synthesizable description using the preprocessing rules defined in this document.

VPP supports a larger subset of Verilog than FV. To do so, we extended our type checking rules and elaboration semantics to support the additional constructs while maintaining the distinction between values that must be known at elaboration time and those that are not. We were able to extend the same two-level approach to the larger subset. The complexity of the concrete syntax caused several engineering problems. For example, the type and direction of a module port can be specified inside the body of a module instead of in the module declaration. Initializing the ports' directions and types to unknown values in the typing environment and updating them while traversing the parsed syntax tree was a simple solution to this problem. VPP also supports behavioral constructs, but does not provide guarantees about their synthesizability since this is implementation dependent.

6.2 Abstractions in Practice

We manually re-factored several industrial hardware descriptions from [11, 13] to use generative constructs. The manual re-factoring was straightforward and mainly involved replacing blocks of instances with loops or nested loops. The hardest part was to figure out the increment expressions for each of the loop indices. Comparing the re-factored code to the original shows that using abstraction can cut the number of lines in half, as depicted in Table 1. Of course, being multipliers, these circuits are highly regular and therefore are particularly suitable to show the usefulness of the abstractions we are talking about. But designing multipliers is still a formidable engineering challenge where engineers use all the help they can get to make the task more tractable.

The results included here are a preliminary experimental results that demonstrates that using abstractions is valuable in practical examples not only for the simple circuits such as ripple adders presented earlier. These results can be significantly improved by using higher levels of abstraction such as type abstraction (currently not supported by Verilog)

6.3 Performance Evaluation

VPP's type checker is based on the type system defined in section 4. As such it is a modular checker and therefore is expected to be reasonably efficient. Table 2 shows the time required to type check the Verilog examples mentioned above along with the code size for each example. It also shows the time required to elaborate the re-factored versions. These experiments were conducted on a

Table 1 Impact of abstraction on code size (in lines).

Circuit	Original	Using Abstractions	Percentage Saved
OpenRISC 1200's 32x32 multiplier [13]	2538	1405	44.6%
OpenSPARC T1's 64x64 multiplier [11]	2510	1167	53.5%

machine with the following specifications: MacBook running Mac OS X version 10.5.6, 2 GHz Intel Core 2 Duo, 4 MB L2 cache, 2 GB 667 MHz DDR2 SDRAM.

As shown in the table, VPP is capable of type checking circuits at a reasonably high rate. Elaboration takes a bit longer as a new abstract syntax tree is constructed for the preprocessed circuit. In general we did not encounter any performance problems with the examples we studied so far.

Table 2 VPP's performance

Circuit	Code Size (lines)	TC (msec)	TC rate (lines/msec)	Elaboration (msec)
Original 32x32 mult.	2538	19.4	130.8	-
Original 64x64 mult.	2510	25.8	97.2	-
Re-factored 32x32 mult.	1405	8.3	169.2	643
Re-factored 64x64 mult.	1167	8.7	134.1	1237

§7 Conclusion

This paper has argued the pressing need for expressive, well-defined preprocessing constructs in hardware description languages, and showed that a hardware description language with such constructs can be understood as a statically typed two-level language. We focused on one of the most basic properties of a circuit description, namely that it corresponds to a synthesizable circuit. We presented Featherweight Verilog (FV), a core calculus (syntax, type system, and preprocessing semantics) that shows how preprocessing constructs can be defined and reasoned about in the context of a revision of a mainstream hardware description language (Verilog). We formalized three technical properties

that capture the key features of our calculus, and imply that well-typed FV programs can be successfully elaborated into well-typed, obviously synthesizable circuit descriptions.

In future work, we intend to address the limitations of our current type system. Namely, we will provide a way to deal with loop termination and inconsistent wire assignments. We believe that more systematic support for elaboration combined with more powerful static checking (before elaboration) can reduce the time, and therefore the cost, needed to produce large scale designs. We believe that we can use such systems to enforce bounds on hardware resources such as area, power and delay. Our long term goal is to demonstrate this thesis in the context of a practical extension to the Verilog language.

Acknowledgment 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, 1984.

2) Bluespec, Inc. *Bluespec SystemVerilog Version 3.8 Reference Guide*, 2006.

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

4) IEEE Standards Board. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. Number 1364-1995 in IEEE Standards. IEEE, 1995.

5) IEEE Standards Board. *IEEE Standard Verilog Hardware Description Language*. Number 1364-2001 in IEEE Standards. IEEE, 2001.

6) IEEE Standards Board. *IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language*. Number 1800-2005 in IEEE Standards. IEEE, 2005.

7) IEEE Standards Board. *IEEE Standard for Verilog Hardware Description Language*. Number 1364-2005 in IEEE Standards. IEEE, 2005.

8) IEEE Standards Board. *IEEE Standard for Verilog Register Transfer Level Synthesis*. Number 1364.1-2002 (IEC 62142:2005) in IEEE Standards. IEEE, 2005.

9) Oleg Kiselyov, Kedar Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *the International Workshop on Embedded Software (EMSOFT '04)*, LNCS, Pisa, Italy, 2004. ACM.

10) Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. *ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

11) Sun Microsystems. Opensparc t1 processor file: mul64.v. `http://opensparc-t1.sunsource.net/nonav/source/verilog/html/mul64.v`.

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

13) Opencores.org. Or1200's 32x32 multiply for asic. `http://www.opencores.org/cvsweb.shtml/or1k/or1200/rtl/verilog/or1200_amultp2_32x32.v`.

14) 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`.

15) Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. available from [14].

16) Walid Taha, Stephan Ellner, and Hongwei Xi. Generating imperative, heap-bounded programs in a functional setting. In *Proceedings of the Third International Conference on Embedded Software*, Philadelphia, PA, 2003.

17) 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.

18) Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

§1 Substitution

A principal challenge in designing preprocessing systems is the avoidance of accidental variable capture, which occurs when the binding occurrence of a variable is changed. Systems such as the C preprocessor (cpp) are seen as fragile because they do not avoid accidental capture. Preprocessing systems that avoid accidental variable capture are called *hygienic* [10]. The key to hygienic preprocessing is to employ a notion of substitution that respects the binding structure of variables – using, for example, free and bound variable conventions as in the lambda calculus [1].

In FV, preprocessing only substitutes level 0 variables (parameters and `for`-loop indices) with their corresponding integral values. Since values are distinct from variables, we do not need to worry about accidental variable capture. Therefore, defining the substitution rules as shown in Figure 4 is a straightforward process. Additionally, since the Barendregt convention is not hiding any complexity, the implementation follows naturally. The only thing to be aware of is the distinction between level 1 variables that should not be affected by the

substitution and level 0 variables that might be. This distinction can easily be made by recording the level of each variable in the abstract syntax tree while traversing the description.

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

$$
\begin{aligned}
& m\langle e_i\rangle\langle l_i\rangle[x \mapsto v] \\
& \qquad = m\langle e_i[x \mapsto v]\rangle\langle l_i[x \mapsto v]\rangle \\
& (\texttt{assign}\ l\ e)[x \mapsto v] \\
& \qquad = \texttt{assign}\ l[x \mapsto v]\ e[x \mapsto v] \\
& (\texttt{if}\ e\ \texttt{then}\ \langle P_i\rangle\ \texttt{else}\ \langle P_j\rangle)[x \mapsto v] \\
& \qquad = \texttt{if}\ e[x \mapsto v]\ \texttt{then}\ \langle P_i[x \mapsto v]\rangle\ \texttt{else}\ \langle P_j[x \mapsto v]\rangle \\
& (\texttt{for}(y = e_1;\ e_2;\ y = e_3)\ \langle P_i\rangle)[x \mapsto v] \\
& \qquad = \texttt{for}(y = e_1[x \mapsto v];\ e_2[x \mapsto v];\ y = e_3[x \mapsto v])\ \langle P_i[x \mapsto v]\rangle
\end{aligned}
$$

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

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

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

Fig. 4 Substitution

§2 Preprocessing Errors

Defining abnormal cases is equally important. Figures 5 and 6 show when errors can occur during preprocessing. Namely, when any of the following situations occur:

1. Elaborating a circuit description:
   a. The main module is not defined (E-PE1).
   b. An error occurs while elaborating its body given all other module definitions (E-PE2).
2. Elaborating the main module body:
   a. The main module has a non-empty parameter sequence (E-BE1).
   b. Elaborating any of the parallel statements in the body generates an error (E-BE2).
3. Elaborating a module instantiation:
   a. Any expression passed as a parameter fails to evaluate (E-ME1).

   b. Any LHS expression passed as a port connection fails to elaborate (E-ME2).
   c. There is no corresponding module definition (E-ME3).
   d. The number of parameters passed is not correct (E-ME4).
   e. The number signals passed is not correct (E-ME5).
   f. An error occurs while elaborating any of the parallel modules composing the body of the instantiated module (E-ME6).

4. Elaborating an assignment:

   a. An error occurs while elaborating its left hand side (E-AssignE1).
   b. An error occurs while elaborating its right hand side (E-AssignE2).

5. Elaborating a conditional statement:

   a. The conditional expression cannot be evaluated (E-IfE).
   b. The conditional expression evaluates to true and any of the parallel statements of the consequent cannot be elaborated (E-IfTrueE).
   c. The conditional expression evaluates to false and any of the parallel statements of the alternative cannot be elaborated (E-IfFalseE).

6. Elaborating loops:

   a. The initialization expression fails to evaluate (E-ForE1).
   b. The condition expression fails to evaluate (E-ForE2).
   c. Any parallel statement in the body of the loop fails to elaborate (E-ForE3).
   d. The remaining loop iterations fail to elaborate (E-ForE4).

7. Elaborating an expression:

   a. It is a signal indexed by one or more expressions that fail to evaluate (E-Idx1E and E-Rg1E).
   b. It is a parameter name (E-Par1E).
   c. It is composed of operations on expressions where at least one fails to elaborate (E-Op1E).

8. Evaluating an expression:

   a. It is a signal (E-IdE, E-Idx0E and E-Rg0E).
   b. It is a parameter name (E-Par0E).
   c. It is composed of operations on expressions including at least one

that fails to evaluate (E-Op0E).

It is important to note that, because we use substitution to eliminate level 0 variables, encountering any identifier during evaluation constitutes a preprocessing error. This formalizes the property that any dependency on either an uninstantiated parameter or a wire value constitutes a preprocessing error.

§3 Proofs and Auxiliary Results

3.1 Substitution Lemma

We only need to define substitution on parallel statements. Therefore we state the substitution lemma as follows:

Lemma 3.1 (Substitution)
If $\Delta; \Gamma, x : d\ t \vdash P$ and $\Gamma \vdash^n v : d\ t$ then $\Delta; \Gamma \vdash P[x \mapsto v]$

Proof Since we only substitute level 0 variables with integral values, we only need to consider the typing rules T-Par and T-Int. A level 0 variable always has type `{in} int` provided that it exists in the environment. Integral values always have the same type `{in} int` unconditionally. As such substitution of a level 0 variable with an integer always preserves the type of the term in which the substitution occurs. ■

We also need to establish the same property for smaller constructs. We omit the proofs of these auxiliary lemmas since they are identical to the previous one. We only need to state Lemmas 3.2, and 3.3 that establish that substitution preserves typing for e and l respectively.

Lemma 3.2 (Substitution for expressions)
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 3.3 (Substitution for LHS values)
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 $\Gamma \vdash^{n_2} l[x \mapsto v] : d_2\ t_2$

3.2 Proof of Type Preservation

Lemmas 3.4, 3.5, 3.6, 3.7, and 3.8 establish that preprocessing preserves typability for e (2 lemmas), l, P, and b terms. They are all required to prove Theorem 5.1

Lemma 3.4 (Type preservation for level 0 expressions)
If $\Gamma^+ \vdash^0 e : d\ t$ and $e \overset{0}{\hookrightarrow} e'$ then $\Gamma^+ \vdash^0 e' : d\ t$

Proof We proceed by induction on the evaluation tree of e. We have six cases: Cases Id, Index, Range are vacuously true because none of them can be typed at level 0. T-Param is also vacuously true because Γ^+ by definition does not contain any level 0 variables. In case Int, the lemma holds trivially from T-Int and E-Int0. Case Op also holds by using the induction hypothesis on e_i in E-Op0 and T-Op. We also assume that the type signature Σ returns by definition the same type that is returned when applying the operator f at level 0. ■

Lemma 3.5 (Type preservation for level 1 expressions)
If $\Gamma^+ \vdash^1 e : d\ t$ and $e \overset{1}{\hookrightarrow} e'$ then $\Gamma^+ \vdash^1 e' : d\ t$

Proof We proceed by induction on the evaluation tree of e. We have six cases:

- **Case Id:** e is s and from E-Id1 e' is s. Therefore the lemma holds trivially.
- **Case Index:** e is $s[e_1]$ and from E-Index1 e' is $s[v]$ where $e_1 \overset{0}{\hookrightarrow} v$. From T-Index we know that $\Gamma^+ \vdash^0 e_1 : \{\texttt{in}\}\ \texttt{int}$. From lemma 3.4, we get that $\Gamma^+ \vdash^0 v : \{\texttt{in}\}\ \texttt{int}$. Using T-index again we get $\Gamma^+ \vdash^1 s[v] : d\ t$.
- **Case Range:** e is $s[e_1 : e_2]$ and from E-Range1 e' is $s[v_1 : v_2]$. Following the same steps of the previous case but using T-Range instead of T-Index and using lemma 3.4 twice we get the desired result.
- **Case Param:** e is x. The lemma hold vacuously because from T-Param it is clear that e cannot be typed using Γ^+.
- **Case Int:** e is v and from E-Int1 e' is v. Therefore the lemma holds trivially.
- **Case Op:** e is $f\langle e_i \rangle$ and from E-Op1 e' is $f\langle e_i' \rangle$. From T-Op we know that $\{\Gamma^+ \vdash^1 e_i : d'\ t_i\}$ and using the induction hypothesis on e_i we get $\{\Gamma^+ \vdash^1 e_i' : d'\ t_i\}$. Using T-Op again, we obtain the required result.

■

Lemma 3.6 (Type preservation for LHS values)
If $\Gamma^+ \vdash^1 l : d\ t$ and $l \overset{1}{\hookrightarrow} l'$ then $\Gamma^+ \vdash^1 l' : d\ t$

Proof This follows directly from Lemma 3.5 because any l is also an e. ■

Lemma 3.7 (Type preservation for parallel statements)
If $\Delta;\Gamma^+ \vdash P$ and $\langle D_i\rangle \vdash P \overset{1}{\hookrightarrow} \langle P'_j\rangle, \langle D_k\rangle$ then $\vdash \langle D_k\rangle : \Delta'$ and $\{\Delta';\Gamma^+ \vdash P'_j\}$

Proof We proceed by induction on the evaluation tree of P. We have four cases:

- **Case Module Instantiation:** From E-Mod, we know that preprocessing P gives a sequence of parallel statements composed of one parallel statement P' equal to $m'\langle\rangle\langle l_j\rangle$ and a set of modules $\langle \texttt{module}\ m'\ \langle\rangle\ \langle d_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle \biguplus_k \langle P_r\rangle^{r\in R(k)}\rangle \uplus \biguplus_k \langle D_h\rangle^{h\in H(k)}$. The preprocessing of each $P_k\{[x_i \mapsto v_i]\}$ is a subtree in the preprocessing of P and since we know from T-Mod that $\Delta(m) = |\langle e_i\rangle|\langle d_j\rangle$, we also know that m is typed which implies from T-Body that every P in $\langle P_k\rangle$ is also typed. Therefore we know from Lemma 3.1 that $P_k\{[x_i \mapsto v_i]\}$ is also typed. From the induction hypothesis we know that all parallel statements $\biguplus_k \langle P_r\rangle^{r\in R(k)}$ and module definitions $\biguplus_k \langle D_h\rangle^{h\in H(k)}$ returned from preprocessing $\langle P_k\rangle$ after substitution are well-typed. More precisely each generated parallel statement P_r is typed under a new module environment containing the types of the modules generated during the preprocessing of the corresponding parallel statement. The final set of modules being the union of all the generated modules in addition to the instantiated module m' is therefore well-typed under the empty environment. The resulting module environment Δ' contains the type of m' in addition to the types of all the generated modules. From T-Mod, we can easily get that $m'\langle\rangle\langle l_j\rangle$ is well-typed under Δ'.
- **Case Assign:** From E-Assign, we know that preprocessing P gives a sequence of new parallel statements composed of one parallel statement P' equal to $\texttt{assign}\ l'\ e'$ and an empty set of modules. From T-Assign we know that both l and e are well-typed under Γ^+ and using Lemmas 3.6 and 3.5 we can conclude that l' and e' have the same types under Γ^+ therefore using T-Assign again we reach $\Delta;\Gamma^+ \vdash \texttt{assign}\ l'\ e'$. From T-MEmpty, we know that the generated set of modules is also well-typed.
- **Case If:** We have two subcases to consider here:
  - **Case True:** From E-IfTrue, we know that preprocessing P gives a sequence of new parallel statements $\biguplus_k \langle P_r\rangle^{r\in R(k)}$, and a sequence of

modules $\biguplus_k \langle D_h \rangle^{h \in H(k)}$. From T-If, we know that $\{\Delta; \Gamma^+ \vdash P_k\}$ and since each parallel statement in P_k is smaller than P, we can apply the induction hypothesis and conclude that all the parallel statements and the modules generated from the preprocessing of each of them are well-typed.

- **Case False:** Similarly we can show that all parallel statements and modules generated from the preprocessing of each parallel statement in $\langle P_j \rangle$ are well-typed.

- **Case For:** We have two subcases to consider here:
  - **Case False:** From E-ForFalse, we know that preprocessing P gives an empty sequence of parallel statements and an empty set of modules. So the result is trivially well-typed.
  - **Case True:** From E-ForTrue, we know that preprocessing P gives a sequence of new parallel statements $\biguplus_k \langle P_r \rangle^{r \in R(k)} \uplus \langle P_j \rangle$, and a sequence of modules $\biguplus_k \langle D_h \rangle^{h \in H(k)} \uplus \langle D_q \rangle$ These sequences are generated from preprocessing the body of the for loop after substitution and preprocessing another `for` statement. First we know from T-For and Lemma 3.4 that $\Gamma^+ \vdash^0 v_1 : \{\texttt{in}\}\ \texttt{int}$. We also know that $\{\Delta; \Gamma^+, y : \{\texttt{in}\}\ \texttt{int} \vdash P_k\}$ and that $\Gamma^+, y : \{\texttt{in}\}\ \texttt{int} \vdash^0 e_2, e_3 : \{\texttt{in}\}\texttt{int}$. Using Lemmas 3.1 and 3.2 respectively we get $\{\Delta; \Gamma^+ \vdash P_k[y \mapsto v1]\}$ and that $\Gamma^+ \vdash^0 e_2[y \mapsto v1], e_3[y \mapsto v1] : \{\texttt{in}\}\texttt{int}$. Now using Lemma 3.4 we know that $\Gamma^+ \vdash^0 v_2 : \{\texttt{in}\}\texttt{int}$. and using the induction hypothesis we get $\vdash \biguplus_k \langle D_h \rangle^{h \in H(k)} : \Delta_1$ and $\forall k. \Delta_1; \Gamma^+ \vdash \langle P_r \rangle^{r \in R(k)}$. Finally using T-For again we get that $\{\Delta; \Gamma^+ \vdash \texttt{for}(y = e_3[y \mapsto v1];\ e_2;\ y = e_3)\ P_k\}$ which by the induction hypothesis shows that $\vdash \langle D_q \rangle : \Delta_2$ and $\forall j. \Delta_2; \Gamma^+ \vdash P_j$. Since all the resulting parallel statements and modules are well-typed then we have the required result.

■

Lemma 3.8 (Type preservation for main module body)
If $\Delta \vdash b : M$ and $\langle D_i \rangle \vdash b \overset{1}{\hookrightarrow} b', \langle D_k \rangle$ then $\vdash \langle D_k \rangle : \Delta'$ and $\Delta' \vdash b' : M$

Proof From E-Body, we can see that the result from preprocessing the body

b of the main module gives a new body b' equal to $\langle\rangle\langle d_j\ s_j\rangle$ `is` $\langle t_q\ s_q\rangle\ \biguplus_k \langle P_r\rangle^{r\in R(k)}$ and a set of modules $\biguplus_k \langle D_h\rangle^{h\in H(k)}$. We also know from T-Body that each parallel statement P in P_k is typed under Δ and $\langle s_j : d_j\ \texttt{wire}\rangle \uplus \langle s_k : \{\texttt{in}, \texttt{out}\}\ t_k\rangle$, therefore using Lemma 3.7 we can conclude that $\biguplus_k \langle P_r\rangle^{r\in R(k)}$ and $\biguplus_k \langle D_h\rangle^{h\in H(k)}$ are well-typed proving the required result. ■

We are now ready to prove the Type Preservation Theorem (Theorem 5.1). The proof works as follows:

Proof Let $p = \langle D_i\rangle m$. From E-Prog, p' is $\langle D_r\rangle \uplus \langle$`module` $m\ b'\rangle\, m$. We also know that $\langle D_i\rangle \vdash b \overset{1}{\hookrightarrow} b', \langle D_r\rangle$ where b is the body of the main module of p. From T-Prog, we know that for p to be typed, all $\langle D_i\rangle$ must be typed and we know from T-MSeq that for this to happen every module's body must be typed under the environment containing the types of modules preceding it. This applies to the main module too. Therefore we can conclude that $\Delta \vdash b : M$ where Δ is the module environment holding the module types of all modules preceding the definition of the main module of p. Given that and using Lemma 3.8 we can conclude that $\vdash \langle D_r\rangle : \Delta'$ and $\Delta' \vdash b' : M$. This also means that $\langle D_r\rangle \uplus \langle$`module` $m\ b'\rangle$ is well-typed under the empty environment (T-MSeq). This proves that $\vdash p'$.

■

3.3 Proof of Preprocessing Soundness

Lemmas 3.9, 3.10, 3.11, 3.12, and 3.13 establish that the result of preprocessing e (2 lemmas), l, P, and b terms respectively produces a fully preprocessed term of the same category. Theorem 5.3 establishes this property for circuit descriptions.

In all the following proofs we just skip the cases where the preprocessing returns `err` since they are trivially in $\hat{X}$.

Lemma 3.9 (Preprocessing soundness for level 0 expressions)
If $e \overset{0}{\hookrightarrow} e'$ then $e' \in \hat{e}$

Proof We proceed by induction on the evaluation tree of e. E-Int0 case is immediate since $e' = v$ which is allowed in $\hat{e}$. Case E-Op0 is also true because

from the induction hypothesis each v_i is in $\hat{e}$ and therefore the result of applying an operator f on $\langle v_i \rangle$ will be in $\hat{e}$. ■

Lemma 3.10 (Preprocessing soundness for level 1 expressions)
If $e \overset{1}{\hookrightarrow} e'$ then $e' \in \hat{e}$

Proof We proceed by induction on the evaluation tree of e. E-Id1 case is immediate since $e' = s$ which is allowed in $\hat{e}$. Cases E-Index1, E-Range1 make use of Lemma 3.9 to make sure that $e' = s[v]$ and $e' = s[v_1 : v_2]$ are in $\hat{e}$. Case E-Int1 is immediate. Finally, case E-Op1 makes use of the induction hypothesis to prove its goal. ■

Lemma 3.11 (Preprocessing soundness for LHS values)
If $l \overset{1}{\hookrightarrow} l'$ then $l' \in \hat{l}$

Proof Follows directly from Lemma 3.10 ■

Lemma 3.12 (Preprocessing soundness for parallel statements)
If $\langle D_i \rangle \vdash P \overset{1}{\hookrightarrow} \langle P'_j \rangle, \langle D_k \rangle$ then $\{P'_j \in \hat{P}\}$ and $\{D_k \in \hat{D}\}$

Proof We use induction on the evaluation tree of P. We have four distinct cases:

- **Case Module Instantiation:** From E-Mod, we know that preprocessing P gives a sequence of parallel statements composed of one parallel statement P' equal to $m'\langle\rangle\langle l_j \rangle$ and a set of modules $\langle \texttt{module}\ m'\ \langle\rangle\ \langle d_j\ s_j \rangle\ \texttt{is}\ \langle t_q\ s_q \rangle \biguplus_k \langle P_r \rangle^{r \in R(k)} \rangle \uplus \biguplus_k \langle D_h \rangle^{h \in H(k)}$ P' is clearly in $\hat{P}$ while the generated set of modules is in $\hat{D}$ if all the parallel statements composing their bodies are in $\hat{P}$. The preprocessing of each $P_k\{[x_i \mapsto v_i]\}$ is a subtree in the preprocessing of P so we know from the induction hypothesis that each of them results in a sequence of parallel statements in $\hat{P}$. Similarly, any modules produced in the process are in $\hat{D}$ by the inductive hypothesis. Note that we know that every v_i is in $\hat{e}$ from Lemma 3.9.
- **Case Assign:** From E-Assign, we know that preprocessing P gives a sequence of parallel statements composed of one parallel statement P' equal to $\texttt{assign}\ l'\ e'$ and an empty set of modules. Using Lemmas 3.11 and 3.10 we obtain that P' is in $\hat{P}$. Trivially the empty set of modules is valid second stage syntax, which concludes the case.
- **Case If:** We have two subcases to consider here:

- **Case True:** From E-IfTrue, The preprocessing result is equal to a sequence of parallel statements and a sequence of modules. These sequences are generated from preprocessing each of the parallel statements in $\langle P_k \rangle$. Since these are parallel statements having smaller evaluation trees than P, we can use the induction hypothesis to show that the results are sound.
- **Case False:** Similarly it can be seen from E-IfFalse, that preprocessing each parallel statement in $\langle P_j \rangle$ produces fully preprocessed terms.

• **Case For:** We have two subcases to consider here:

- **Case False:** From E-ForFalse, The preprocessing result is equal to an empty sequence of parallel statements and an empty sequence of modules.
- **Case True:** From E-ForTrue, The preprocessing result is equal to a sequence of parallel statements and a sequence of modules. This sequence is generated from preprocessing the body of the for loop after substitution and preprocessing another `for` statement. Since both the body of the loop and the new loop are parallel statements having smaller evaluation trees than P, we can use the induction hypothesis to show that the results are sound. Again, note that substitutions in E-ForTrue do not introduce any subterms that are not in our preprocessed syntax since we are substituting using v_1 which is obtained by evaluating e_1 at level 0 and is therefore in $\hat{e}$ as proved in Lemma 3.9.

■

Lemma 3.13 (Preprocessing soundness for main module body)
If $\langle D_i \rangle \vdash b \xhookrightarrow{1} b', \langle D_k \rangle$ then $b' \in \hat{b}$ and $\{D_k \in \hat{D}\}$

Proof From E-Body, we can see that the result from preprocessing the body b of the main module gives a new body b' equal to $\langle\rangle\langle d_j\ s_j\rangle$ `is` $\langle t_q\ s_q\rangle \biguplus_k \langle P_r \rangle^{r \in R(k)}$ and a set of modules $\biguplus_k \langle D_h \rangle^{h \in H(k)}$. Note that b' is parameter free and that from Lemma 3.12 we know that all parallel statements used to construct $\biguplus_k \langle P_r \rangle^{r \in R(k)}$ are in $\hat{P}$ and that all module definitions used to construct $\biguplus_k \langle D_h \rangle^{h \in H(k)}$ are in $\hat{D}$ because they all come from parallel statements preprocessing. Therefore $b' \in \hat{b}$ and $\{D_k \in \hat{D}\}$ are true. ■

Finally, we prove the Preprocessing Soundness Theorem (Theorem 5.3) as follows:

Proof From E-Prog, p' is $\langle D_r \rangle \uplus \langle \texttt{module}\ m\ b' \rangle\, m$ where $\langle D_i \rangle \vdash b \overset{1}{\hookrightarrow} b', \langle D_r \rangle$ and b is the body of the main module of p. From Lemma 3.13 we know that for every D_r, we have $D_r \in \hat{D}$ and that $b' \in \hat{b}$, therefore p' is in $\hat{p}$. ■

$$\boxed{\langle D\rangle \vdash P \overset{1}{\hookrightarrow} \langle P_\perp\rangle, \langle D\rangle} \qquad \frac{\exists i.e_i \overset{0}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ME1)}$$

$$\frac{\exists j.l_i \overset{1}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ME2)} \qquad \frac{\texttt{module}\ m\ \langle x_i\rangle\ \langle d_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \notin \langle D_i\rangle}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ME3)}$$

$$\frac{\texttt{module}\ m\ \langle x_r\rangle\ \langle d_h\ s_h\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \in \langle D_i\rangle \qquad |\,\langle e_i\rangle\,| \neq |\,\langle x_r\rangle\,|}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ME4)}$$

$$\frac{\texttt{module}\ m\ \langle x_i\rangle\ \langle d_h\ s_h\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \in \langle D_i\rangle \qquad |\,\langle l_j\rangle\,| \neq |\,\langle d_h\ s_h\rangle\,|}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ME5)}$$

$$\frac{\begin{array}{l}\texttt{module}\ m\ \langle x_i\rangle\ \langle d_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \in \langle D_i\rangle \\ \{e_i \overset{0}{\hookrightarrow} v_i\} \qquad \exists k.\langle D_i\rangle \vdash P_k\{[x_i \mapsto v_i]\} \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle\end{array}}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ME6)}$$

$$\frac{l \overset{1}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash \texttt{assign}\ l\ e \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-AssignE1)} \qquad \frac{e \overset{1}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash \texttt{assign}\ l\ e \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-AssignE2)}$$

$$\frac{e \overset{0}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash \texttt{if}\ e\ \texttt{then}\ \langle P_k\rangle\ \texttt{else}\ \langle P_j\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-IfE)}$$

$$\frac{e \overset{0}{\hookrightarrow} v \qquad v \neq 0^{32} \qquad \exists k.\langle D_i\rangle \vdash P_k \overset{1}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash \texttt{if}\ e\ \texttt{then}\ \langle P_k\rangle\ \texttt{else}\ \langle P_j\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-IfTrueE)}$$

$$\frac{e \overset{0}{\hookrightarrow} 0^{32} \qquad \exists j.\langle D_i\rangle \vdash P_j \overset{1}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash \texttt{if}\ e\ \texttt{then}\ \langle P_k\rangle\ \texttt{else}\ \langle P_j\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-IfFalseE)}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash \texttt{for}(y = e_1;\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ForE1)}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v_1 \qquad e_2[y \mapsto v_1] \overset{0}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash \texttt{for}(y = e_1;\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ForE2)}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v_1 \qquad \exists k.\langle D_i\rangle \vdash P_k[y \mapsto v_1] \overset{1}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash \texttt{for}(y = e_1;\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ForE3)}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v_1 \qquad \langle D_i\rangle \vdash \texttt{for}(y = e_3[y \mapsto v_1];\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}{\langle D_i\rangle \vdash \texttt{for}(y = e_1;\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle \texttt{err}\rangle, \langle\rangle}\ \text{(E-ForE4)}$$

Fig. 5 Operational Semantics Errors For Parallel Statements

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

$$\frac{\texttt{module}\ m\ b \notin \langle D_i \rangle}{\langle D_i \rangle\ m \overset{1}{\hookrightarrow} \texttt{err}}\ \text{(E-PE1)} \qquad \frac{\texttt{module}\ m\ b \in \langle D_i \rangle \quad \langle D_i \rangle \vdash b \overset{1}{\hookrightarrow} \texttt{err}, \langle\rangle}{\langle D_i \rangle\ m \overset{1}{\hookrightarrow} \texttt{err}}\ \text{(E-PE2)}$$

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

$$\frac{I \neq \emptyset}{\langle D_i \rangle \vdash \langle x_i \rangle^{i \in I} \langle d_j\ s_j \rangle\ \texttt{is}\ \langle t_q\ s_q \rangle \langle P_k \rangle \overset{1}{\hookrightarrow} \texttt{err}, \langle\rangle}\ \text{(E-BE1)}$$

$$\frac{\exists k. \langle D_i \rangle \vdash P_k \overset{1}{\hookrightarrow} \langle \texttt{err} \rangle, \langle\rangle}{\langle D_i \rangle \vdash \langle\rangle \langle d_j\ s_j \rangle\ \texttt{is}\ \langle t_q\ s_q \rangle \langle P_k \rangle \overset{1}{\hookrightarrow} \texttt{err}, \langle\rangle}\ \text{(E-BE2)}$$

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

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

$$\frac{e \overset{0}{\hookrightarrow} \texttt{err}}{s[e] \overset{1}{\hookrightarrow} \texttt{err}}\ \text{(E-Idx1E)} \qquad \frac{\exists i. e_i \overset{0}{\hookrightarrow} \texttt{err}}{s[e_1 : e_2] \overset{1}{\hookrightarrow} \texttt{err}}\ \text{(E-Rg1E)} \qquad \frac{}{x \overset{1}{\hookrightarrow} \texttt{err}}\ \text{(E-Par1E)}$$

$$\frac{\exists i. e_i \overset{1}{\hookrightarrow} \texttt{err}}{f \langle e_i \rangle \overset{1}{\hookrightarrow} \texttt{err}}\ \text{(E-Op1E)}$$

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

$$\frac{}{s \overset{0}{\hookrightarrow} \texttt{err}}\ \text{(E-IdE)} \qquad \frac{}{s[e] \overset{0}{\hookrightarrow} \texttt{err}}\ \text{(E-Idx0E)} \qquad \frac{}{s[e_1 : e_2] \overset{0}{\hookrightarrow} \texttt{err}}\ \text{(E-Rg0E)}$$

$$\frac{}{x \overset{0}{\hookrightarrow} \texttt{err}}\ \text{(E-Par0E)} \qquad \frac{\exists i. e_i \overset{0}{\hookrightarrow} \texttt{err}}{f \langle e_i \rangle \overset{0}{\hookrightarrow} \texttt{err}}\ \text{(E-Op0E)}$$

Fig. 6 Operational Semantics Errors For All Other Constructs