Synthesizable High Level Hardware Descriptions [*]

Using Statically Typed Two-Level Languages to Guarantee Verilog Synthesizability

Jennifer Gillenwater Gregory Malecha
Cherif Salama Angela Yun Zhu
Walid Taha

Rice University
{jgillenw,gmalecha,cherif,angela.zhu,taha}@rice.edu

Jim Grundy John O'Leary

Intel Strategic CAD Labs
{jim.d.grundy,john.w.oleary}@intel.com

Abstract

Modern hardware description languages support code-generation constructs like `generate/endgenerate` in Verilog. These constructs are intended 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 constructs 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. By viewing Verilog as a statically typed two-level language, we are able 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 abstractions such as iteration and module parameters are used in a synthesizable manner. To illustrate this idea, 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 expansion is always a synthesizable circuit.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation; Preprocessors; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages, Standardization, Theory, Verification

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

[*] This work was supported by the National Science Foundation (NSF) SoD award 0439017, and the Semiconductor Research Consortium (SRC) Task ID: 1403.001 (Intel custom project).

1. Introduction

The Verilog language has three kinds of constructs. Constructs of the first kind are used to describe synthesizable hardware components. These are usually referred to as *structural* constructs. Constructs of the second kind describe behaviors in a simulation environment and are ignored during synthesis. These allow the description of circuit functionality at an algorithmic level, 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*. This kind includes parameterized modules, conditionals, and iterations and is important because it allows for a clear and compact description of regular designs as well as reusable descriptions of circuit families.

For example, for-loops and module parameterization can be used to describe a family of ripple adders 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
```

This example uses a sequence of full adder instances to construct a ripple adder of variable size `N`. In this specific case, the parameter `N` is assigned the default value 4 which means that when the module is instantiated, a 4-bit ripple adder will be created by default. A different value can be specified when instantiating the module to create differently-sized ripple adders. That is why it is said that the above code describes a family of ripple adders and not a specific one. This way, parameterized modules allow designers to write generic circuit descriptions.

The line between synthesizable and non-synthesizable descriptions is unclear and *ad hoc*. The Verilog Register Transfer Language (RTL) synthesis standard (IEEE 1364.1 [9]) does not formally define synthesizable descriptions. Instead, it gives synthesizability guidelines, illustrated by a series of 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 [5]) 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 [6], and subsequent Verilog-2005 [8] and System Verilog [7] standards, extend Verilog-95 with the `generate/endgenerate` construct, which allows conditionals and single variable iteration statements to appear in parallel statements. In the context of `generate` blocks, these statements are elaborated into ordinary parallel statements prior to simulation or synthesis. Note that the use of the keywords `generate` and `endgenerate` is optional. Any conditional or iterative parallel statement is implicitly interpreted as being inside a `generate` block.

Because `generate` blocks are analyzed only after they are elaborated into more Verilog code, errors remain undetected until synthesis, when they are more difficult to analyze. In other words, 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 expanded 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]);
```

As an alternative to writing such descriptions manually, it is not unusual to see designers using scripting languages, such as Perl, to generate Verilog code for specific instances of modules families. Both unrolling and scripting are far from ideal. The first is tedious, error-prone and defeats many principles of software engineering. The second uses a language that has no understanding of the hardware description it generates. It treats everything as a set of strings rendering static analysis of any kind impractical. The generated code is not even guaranteed to be syntactically correct! [1]

It does not have to be this way. Two-level [4, 13] and multi-level languages [16, 18] 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. Kiselyov, Swadi and Taha [10] have shown how to generate highly optimized, type correct Fast Fourier Transform kernel routines from compact algorithmic descriptions written using multi-level languages. Taha, Ellner and Xi [17] have shown how to generate heap bounded implementations of sorting programs (essentially, malloc-free C implementations) from a compact, parameterized sorting algorithm written using two-level languages. In both cases all static analysis is performed prior to code generation.

Bluespec SystemVerilog (BSV) also has powerful generate-like features with static checks. However we are not aware of a formal account of the semantics of these constructs or of BSV in general. BSV's reference guide [2] alludes to the semantics of BSV being defined based on Term Rewriting Systems (TRS) [19]. However, the relationship between BSV and TRS is not formally explained. A formal semantics is a prerequisite for having static guarantees about synthesizability.

[1] Emacs also has an advanced Verilog mode that provides macro expansion capabilities. This mode reduces the amount of typed code by inserting AUTOS comments that expand into corresponding Verilog code. This approach is not an alternative to unrolling or scripting because these macro do not support generate-like code, they only infer and insert some missing information that the programmer does not want to type (e.g. port lists, sensitivity lists).

Our thesis is that the techniques developed for statically typed two-level languages are particularly pertinent to hardware description languages. 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. Similar to the way Taha, Ellner and Xi [17] used type systems to characterize software heap bounds, 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 a context of a practical extension to the Verilog language.

1.1 Contributions

Our key contribution is to show that, by treating Verilog as a statically typed two-level language, we can statically check the synthesizability of a description with high-level abstractions without having to elaborate it. Demonstrating this possibility:

- Provides a proof of concept that we can check properties of the circuit generated from elaboration without actually performing elaboration. This paves the way for more aggressive static checking.
- Suggests that designers may be able to use high level abstractions and get all the associated advantages. This can be done without compromising synthesizability because any misuse of abstractions that might impair synthesizability will be detected statically.
- Allows designers to use a single — tightly integrated — language to describe circuits and circuit families.
- Allows us to use the same type checker to check descriptions before and after elaboration.
- Allows for checking the synthesizability of families of circuits once, rather than at each instantiation.
- Saves wasted time elaborating or synthesizing circuits that will fail in either of these phases.

In trying to achieve these goals, we were lead to another contribution that may be of broader significance and applicability: We introduce and rigorously define *obvious synthesizability* and *general synthesizability* (or *synthesizability* for short) in an implementation independent way (Section 2.4). These concepts allow us to pin down the Verilog constructs that are interesting from the synthesizability point of view and formally treat them.

To address these issues from a semantic point of view, we define Featherweight Verilog (FV), a calculus for a representative core of structural Verilog (Section 2). Using ideas from two-level languages, we give a type system to FV that reflects the conditions needed to guarantee that various constructs do not interfere with synthesizability (Section 3). We develop a two-level operational semantics for FV that captures how various constructs should be elaborated as well as what can go wrong during this process (Section 4).

We establish three properties of FV (Section 5). Theorem 1 shows that a well-typed design elaborates to a well-typed design. Theorem 2 shows that it is always safe to perform elaboration of a well-typed design at compile time because elaboration never depends on wire values. Theorem 3 shows that the result of elaboration is a design that is itself free of elaboration constructs. Combined, these results show that a well-typed design is always synthesizable, which implies that we can statically check for synthesizability of a description before elaborating it using a relatively simple type checker. The complete proofs of these theorems are in the extended technical report version of this paper [3].

We also provide a prototype implementation of a Verilog Pre-Processor (VPP). 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

FV focuses on the structural (rather than behavioral) subset because we are interested in synthesizability. The behavioral subset is primary intended for simulation and testing. Therefore, we do not need to model registers. This does not limit our ability to describe sequential circuits since these can be described structurally either by assuming a primitive flip-flop module or by describing one using primitive gates and feedbacks. Moreover, whereas the full Verilog language provides many constructs to make writing hardware descriptions as convenient as possible, a calculus must be minimized to facilitate analysis of the essential features of the language. With this goal in mind, we model only signals, primitive gates, conditionals, iterations, and modules. A module is a circuit with input and output signals. In addition, a module's type can be parameterized by a set of integer-values. Restricting ourselves to this core calculus in our formal presentation does not mean that the ideas we present are only applicable to the presented subset. Our prototype implementation currently supports a larger subset and will eventually grow to support the full language.

2.1 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. $\mathbb{O}$ is the finite set of operator names, and $\mathbb{N}$ is the set of natural numbers. The full grammar for FV is defined as follows:

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

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 is a name m and a module body b. A module body itself consists of fours 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 $\langle t_k\ s_k \rangle^{k \in K}$, 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 `wire` or `int`. 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 $f \langle e_i \rangle^{i \in I}$. The choice of 32-bit values reflects the peculiar manner in which Verilog interprets parameter values when they are viewed as wire signals.

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

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

2.2 Relation of Calculus to Verilog

The calculus closely resembles the concrete syntax for Verilog but has been simplified for convenience. In particular, in FV:

- All sequences are represented uniformly as `<a,b,...,z>`,
- We indicate the start of the module body with the terminal `is`,
- Local variable declarations are aggregated immediately after the `is` terminal,
- Module parameters are not declared locally, but rather, listed separately before the usual formal parameters,
- Module parameters do not have default values and therefore appropriate values must be passed to any parameterized module in order to instantiate it.
- The directions of ports in the formal argument list to a module are declared in the formal argument itself (as allowed starting from Verilog-2001), rather than in the body of the module definition (as in Verilog-95),
- Names of several Verilog keywords, such as `input` and `output`, are replaced by shorter names, such as `in` and `out`,
- Variable direction is represented using a set that can be equal to $\{\texttt{in}\}$, $\{\texttt{out}\}$, or $\{\texttt{in}, \texttt{out}\}$ denoting `in`, `out`, and `inout` respectively. The latter is also used for non-directional variables such as internal signals.
- The if-statement requires an `else` clause. Since the alternative part can be an empty sequence of statements, this is only a cosmetic constraint,
- 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 do not need to be declared explicitly and are local to the loop,
- Primitive gates are not explicitly modeled, but they can be expressed using logical operators, and
- Integers are represented in binary.

2.3 Notational Conventions

We use the following conventions in the formal treatment of FV:

- 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 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 $| \langle X_i \rangle |$ for the length of the sequence $\langle X_i \rangle$.

2.4 Synthesizable Subset

To proceed, we require a deeper understanding of what is synthesizable and what is not in FV. To do so we introduce and define two general concepts:

- **obvious synthesizability** which 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) which 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 different ways to implement a circuit even 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. This also means that descriptions that are obviously synthesizable according to our definition should be synthesizable by all sensible synthesis tools supporting the language in which the description is written.

Applying this definition to Verilog, it is clear that, when free from high level abstractions, well-formed structural Verilog descriptions are obviously synthesizable. The same applies to FV as well since it is a subset of structural Verilog: All well-formed FV descriptions that are abstraction-free (in this case free from 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 the next section.

3. Type System

This section presents a type system for FV which defines synthesizability. We show how to type check `if`-conditions and `for`-loop constructs and we show what preprocessing computations are implicitly embodied in a design that uses abstraction mechanisms such as iterations, conditionals, and module parameters. In the terminology of two-level languages, *preprocessing*[2] is the level 0 computation, and the result after preprocessing is the level 1 computation that is performed by the circuit. In a Verilog description, there are relatively few places where preprocessing is required. In FV, these are restricted to four places: 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.

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 expansion 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. $\vdash^n$.

[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 we hope our tool will enjoy.

3.1 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. An operator signature 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. We do not have to keep level information in the variable environment because we can differentiate between levels syntactically. 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.

3.2 Typing Rules

Figure 1 defines the rules for the judgment $\vdash p$. A circuit description p is well-typed when this judgment is derivable. 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). Intuitively, 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).

The rules T-MEmpty and T-MSeq defines a well-typed module sequence. Intuitively, the body of each module in the sequence must be well-typed and then the module type environment is extended to reflect this module's type.

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 syntacticly distinguishable.

The next set of rules is used to defines a well-typed parallel statement based on its 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 the conditional expression to be 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.

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

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

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

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

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

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

$$\boxed{\Gamma \vdash^n e : d\; t} \qquad \frac{\Gamma(s) = d\; t}{\Gamma \vdash^1 s \;:\; d\; t}\,(\mathrm{T-Id}) \qquad \frac{\begin{array}{l}\Gamma(s) = d\; t \\ \Gamma \vdash^0 e : \{\texttt{in}\}\;\texttt{int}\end{array}}{\Gamma \vdash^1 s[e] : d\; t}\,(\mathrm{T-Idx}) \qquad \frac{\begin{array}{l}\Gamma(s) = d\; t \\ \Gamma \vdash^0 e_1, e_2 : \{\texttt{in}\}\;\texttt{int}\end{array}}{\Gamma \vdash^1 s[e_1 \;:\; e_2] : d\; t}\,(\mathrm{T-Rg})$$

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

Figure 1. Type System

(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 $\{\texttt{in}\}$).

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-Param, T-Int and, T-Op always use $\{\texttt{in}\}$ 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 can be 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.

3.3 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.

4. Operational Semantics for 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 expansion should be performed, what the form of the preprocessed circuit descriptions should be, and what errors can occur during preprocessing.

4.1 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 during expansion. 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* [11]. 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]. Once the notion of substitution is defined correctly, no additional renaming of signals is needed in the operational semantics. The two central issues that must be treated with care are:

- Using the Barendregt convention for variables means assuming that the set of free and bound variables in any meta-theoretic expression are different. This makes the definition of substitution deceptively simple, as it requires that we implicitly perform all renaming necessary to ensure that the condition on a set of variables holds. This in turn requires,
- Being explicit about the difference between free and binding occurrences of variables in FV. Free variables are relatively easy to recognize from the grammar definition. The constructs that bind variables are more subtle, and are module definitions and `for`-loops. Their behavior as binding constructs is reflected in the type system, and in particular by the way they extend the typing environment.

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 2 is a straightforward process. Ad-

$$
\begin{array}{lllll}
\boxed{P[x \mapsto v]} & m\langle e_i\rangle\langle l_i\rangle[x \mapsto v] & = & m\langle e_i[x \mapsto v]\rangle\langle l_i[x \mapsto v]\rangle \\
 & (\texttt{assign}\ l\ e)[x \mapsto v] & = & \texttt{assign}\ l[x \mapsto v]\ e[x \mapsto v] \\
 & (\texttt{if}\ e\ \texttt{then}\ \langle P_i\rangle\ \texttt{else}\ \langle P_j\rangle)[x \mapsto v] & = & \texttt{if}\ e[x \mapsto v]\ \texttt{then}\ \langle P_i[x \mapsto v]\rangle\ \texttt{else}\ \langle P_j[x \mapsto v]\rangle \\
 & (\texttt{for}(y = e_1;\ e_2;\ y = e_3)\ \langle P_i\rangle)[x \mapsto v] & = & \texttt{for}(y = e_1[x \mapsto v];\ e_2[x \mapsto v];\ y = e_3[x \mapsto v])\ \langle P_i[x \mapsto v]\rangle \\
 & & & \\
\boxed{l[x \mapsto v]} & \text{See } e[x \mapsto v] & & \\
 & & & \\
\boxed{e[x \mapsto v]} & s[x \mapsto v] & = & s \\
 & s[e][x \mapsto v] & = & s[e[x \mapsto v]] \\
 & s[e_1 : e_2][x \mapsto v] & = & s[e_1[x \mapsto v] : e_2[x \mapsto v]] \\
 & x[x \mapsto v] & = & v \\
 & y[x \mapsto v] & = & y \qquad \text{if } y \neq x \\
 & v'[x \mapsto v] & = & v' \\
 & f\langle e_i\rangle[x \mapsto v] & = & f\langle e_i[x \mapsto v]\rangle
\end{array}
$$

Figure 2. Substitution

ditionally, 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.

4.2 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.

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 $X_\perp$. When the $\langle D_i\rangle$ or $\langle D_j\rangle$ components are irrelevant to the judgment, they will simply be dropped. $\langle D_i\rangle$ can be dropped when we process an entity that does not require knowledge about the modules available in the context, and $\langle D_j\rangle$ can be omitted when we process an entity that cannot instantiate new modules. The value of n can either be 1, to indicate preprocessing, or 0, to indicate evaluation.

Normal preprocessing is defined in Figure 3. These rules formalize the following: Preprocessing a circuit description (E-Prog) starts by preprocessing the main module; other modules are instantiated as needed. Preprocessing the body of the main module (E-Body) involves preprocessing each of its statements. Preprocessing a statement can involve instantiating several modules. All such 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. Preprocessing the assignment statement (E-Assign) is trivial because all the work is done ahead of time by substitution. The rules for `if` (E-IfTrue and E-IfFalse) are straightforward. Preprocessing a `for`-loop (E-ForTrue and E-ForFalse) 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 that are at level 1 (E-Id, E-Idx, E-Rg, E-Int1, and E-Op1) are preprocessed, and ones at level 0 (E-Int0 and E-Op0) are evaluated normally. For expressions at level 0, the only active rule in evaluation pertains to operator applications (E-Op0).

4.3 Preprocessing Errors

Defining abnormal cases is equally important. Figure 4 shows 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:

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

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

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

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

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

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

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

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

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

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

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

$\boxed{l \overset{1}{\hookrightarrow} l_\perp}$

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

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

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

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

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

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

Figure 3. Operational Semantics

(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 fail 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.

Consider the following example where we have a main module trying to instantiate two ripple adder modules, one of size four and one of size `s1`.

```
module main();
   wire cout1,cout2;
   wire [3:0] a1,b1,a2,b2,s1,s2;

   adder #(4) d1 (s1,cout1,a1,b1,0);
   adder #(s1) d2 (s2,cout2,a2,b2,0);
endmodule
```

This is a typical example of a description that will fail to elaborate. This is easy to see by looking at rule (E-ME1) because `s1` will fail to evaluate as defined by (E-IdE). This failure is expected because trying to instantiate a ripple-adder using `s1` as a parameter is equivalent to trying to create a ripple adder whose size is variable depending on the output value from the first adder. Clearly this cannot be physically realized. According to T-Mod, for this program to be well-typed, s1 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 can successfully detect that this program is not well-typed and therefore not synthesizable.

As we will show in the next section, the type system guarantees that none of these errors can occur.

5. Technical Results

We establish three theorems whose complete proofs are presented in the technical report [3].

5.1 Preprocessing Produces Well-Typed Circuit Descriptions

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

Lemma 1 (Substitution). *If* $\Delta;\Gamma, x : d\ t \vdash P$ *and* $\Gamma \vdash^n v : d\ t$ *then* $\Delta;\Gamma \vdash P[x \mapsto v]$

Sketch. The proof proceeds by induction on the derivation of the first judgment. □

Using this lemma, we show that preprocessing of a well-typed description produces a well-typed description. Formally:

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

Sketch. The proof proceeds by induction on the derivation of the second judgment. □

While this is an important property, it still allows two undesirable behaviors that our type system forbids: First, it is possible for preprocessing to produce the value `err`. Second, it is possible for preprocessing to produce a value p' that is not `err`, but still contains constructs that we want synthesis to eliminate. The next two results address these issues.

5.2 Preprocessing does not Depend on Wire Values (and is Type Safe)

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 2 (Type Safety). *If* $\vdash p$ *and* $p \overset{1}{\hookrightarrow} p'$ *then* $p' \neq$ `err`

Sketch. This result follows directly from Theorem 1 since no typing rules will consider `err` well-typed. □

5.3 Preprocessing Produces Fully Expanded Terms

To show that preprocessing produces fully expanded terms, we must formalize this notion. The set of fully expanded terms is defined as follows:

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

The above definition formally states that for a term to be fully expanded it has to satisfy the following four conditions:

- If it contains a module declaration, the module must not have parameters.
- If it contains a module instantiation, then the instantiation must not pass any parameters.
- It cannot contain `for`-loops.
- It cannot contain `if`-statements.

Note that `err` is considered to be a fully expanded term since it does not include any abstractions.

Theorem 3 establishes the soundness of preprocessing which refers to the property that elaboration produces fully expanded descriptions.

Theorem 3 (Preprocessing Soundness). *If* $p \overset{1}{\hookrightarrow} p'$ *then* $p' \in \hat{p}$

Sketch. The proof proceeds by induction on the derivation of the first judgment. □

As stated in Section 2.4, 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 2 says its elaboration will not produce an error and Theorems 1 and 3 say that the result will be abstraction-free and well-typed. 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 expansion rules defined in this document.

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

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

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

$$\frac{I \neq \emptyset}{\langle D_i\rangle \vdash \langle x_i\rangle^{i\in I}\langle d_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\langle P_k\rangle \overset{1}{\hookrightarrow} \texttt{err}, \langle\rangle}\ (\text{E}-\text{BE1}) \qquad \frac{\exists k.\langle D_i\rangle \vdash P_k \overset{1}{\hookrightarrow} \langle\texttt{err}\rangle, \langle\rangle}{\langle D_i\rangle \vdash \langle\rangle\langle d_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\langle P_k\rangle \overset{1}{\hookrightarrow} \texttt{err}, \langle\rangle}\ (\text{E}-\text{BE2})$$

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

$$\frac{\exists i.e_i \overset{0}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \langle\texttt{err}\rangle, \langle\rangle}\ (\text{E}-\text{ME1}) \qquad \frac{\exists j.l_i \overset{1}{\hookrightarrow} \texttt{err}}{\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \overset{1}{\hookrightarrow} \langle\texttt{err}\rangle, \langle\rangle}\ (\text{E}-\text{ME2})$$

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

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

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

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

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

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

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

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

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

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

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

$$\boxed{l \overset{1}{\hookrightarrow} l_\perp}$$

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

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

$$\frac{e \overset{0}{\hookrightarrow} \texttt{err}}{s[e] \overset{1}{\hookrightarrow} \texttt{err}}\ (\text{E}-\text{Idx1E}) \qquad \frac{\exists i.e_i \overset{0}{\hookrightarrow} \texttt{err}}{s[e_1 : e_2] \overset{1}{\hookrightarrow} \texttt{err}}\ (\text{E}-\text{Rg1E}) \qquad \frac{}{x \overset{1}{\hookrightarrow} \texttt{err}}\ (\text{E}-\text{Par1E}) \qquad \frac{\exists i.e_i \overset{1}{\hookrightarrow} \texttt{err}}{f\langle e_i\rangle \overset{1}{\hookrightarrow} \texttt{err}}\ (\text{E}-\text{Op1E})$$

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

$$\frac{}{s \overset{0}{\hookrightarrow} \texttt{err}}\ (\text{E}-\text{IdE}) \qquad \frac{}{s[e] \overset{0}{\hookrightarrow} \texttt{err}}\ (\text{E}-\text{Idx0E}) \qquad \frac{}{s[e_1 : e_2] \overset{0}{\hookrightarrow} \texttt{err}}\ (\text{E}-\text{Rg0E}) \qquad \frac{}{x \overset{0}{\hookrightarrow} \texttt{err}}\ (\text{E}-\text{Par0E})$$

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

Figure 4. Operational Semantics Errors

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 re-factored several industrial hardware descriptions from [12, 14] to use higher level abstractions. 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 level of abstractions such as type abstractions.

Circuit	Original	Using Abstractions	Percentage Saved
OpenRISC 1200's 32x32 multiplier [14]	2538	1405	44.6%
OpenSPARC T1's 64x64 multiplier [12]	2510	1167	53.5%

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

7. Conclusions and Future Work

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 developed 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 always be successfully elaborated into well-typed, obviously synthesizable circuit descriptions.

In future work, we intend to enrich the underlying type system to capture physical features of the hardware design, including area, timing, and power requirements.

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, 1984.

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

[3] Jennifer Gillenwater, Gregory Malecha, Cherif Salama, Angela Yun Zhu, Walid Taha, Jim Grundy, and John O'Leary. Synthesizable High Level Hardware Descriptions. Technical report, Rice University and Intel Strategic CAD Labs, http://www.resource-aware.org/twiki/pub/RAP/VPP/FV-TR.pdf, 2007.

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

[5] IEEE Standards Board. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. Number 1364-1995 in IEEE Standards. IEEE, 1995.

[6] IEEE Standards Board. *IEEE Standard Verilog Hardware Description Language*. Number 1364-2001 in IEEE Standards. IEEE, 2001.

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

[8] IEEE Standards Board. *IEEE Standard for Verilog Hardware Description Language*. Number 1364-2005 in IEEE Standards. IEEE, 2005.

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

[10] 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.

[11] 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.

[12] Sun Microsystems. Opensparc t1 processor file: mul64.v. http://opensparc-t1.sunsource.net/nonav/source/verilog/html/mul64.v.

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.