HOSC manuscript No.
(will be inserted by the editor)

Static Consistency Checking for Verilog Wire Interconnects

Using Dependent Types to Check the Sanity of Verilog Descriptions

Cherif Salama · Gregory Malecha · Walid Taha · Jim Grundy · John O'Leary

Received: date / Accepted: date

Abstract The Verilog hardware description language has padding semantics that allow designers to write descriptions where wires of different bit widths can be interconnected. However, many such connections are nothing more than bugs inadvertently introduced by the designer and often result in circuits that behave incorrectly or use more resources than required. A similar problem occurs when wires are incorrectly indexed by values (or ranges) that exceed their bounds. These two problems are exacerbated by `generate` blocks. While desirable for reusability and conciseness, the use of `generate` blocks to describe circuit families only makes the situation worse as it hides such inconsistencies. Inconsistencies in the generated code are only exposed after elaboration when the code is fully-expanded.

In this paper we show that these inconsistencies can be pinned down prior to elaboration using static analysis. We combine dependent types and constraint generation to reduce the problem of detecting the aforementioned inconsistencies to a satisfiability problem. Once reduced, the problem can easily be solved with a standard satisfiability modulo theories (SMT) solver. In addition, this technique allows us to detect unreachable code when it resides in a block guarded by an unsatisfiable set of constraints. To illustrate these ideas, we develop a type system for Featherweight Verilog (FV), a core calculus of structural Verilog with generative constructs and previously defined elaboration semantics. We prove that a well-typed FV description will always elaborate into an inconsistency-free description. We also provide an open-source implementation demonstrating our approach.

Keywords Verilog Elaboration · Static Array Bounds Checking · Verilog Wire Width Consistency · Dead Code Elimination · Dependent Types

An earlier version was presented at PEPM'09. This manuscript includes an extended set of examples, performance evaluation, complete proofs, and various improvements. 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).

C. Salama · G. Malecha · W. Taha
Rice University. E-mail: {cherif, gmalecha, taha}@rice.edu

J. Grundy · J. O'Leary
Intel Strategic CAD Labs. E-mail: {jim.d.grundy, john.w.oleary}@intel.com

1 Introduction

Digital circuit design is becoming increasingly complex. Processors nowadays have hundreds of millions of transistors. Hardware description languages (HDLs) like Verilog and VHDL would have been completely useless had they required the designers to describe the circuits they want to build at the transistor level. Thankfully, this is not the case. HDLs provide designers with various kinds of abstractions to be able to describe hardware at higher levels. For example, instead of using transistors, a designer can use gates, flip-flops, latches, or complete hardware modules as circuit building blocks offering a hierarchal approach to hardware design. Productivity can be dramatically increased using higher levels of abstractions, which are therefore very desirable. Verilog [6] currently provides two types of constructs toward this end:

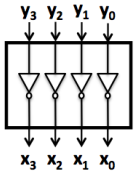
- **Behavioral Constructs**: these constructs allow a designer to describe a circuit functionality in a algorithmic way as one would do in C. The resultant description is suitable for simulation purposes but unless adhering to a rather ad hoc set of guidelines it cannot be synthesized into a hardware circuit. Even when synthesizable, the hardware designer has limited control over the generated hardware.
- **Generative Constructs**: these constructs allow a designer to describe circuit families generically and structurally. These were introduced in the 2001 IEEE Verilog standard [5]. They are particularly useful when describing frequently used modules that can be thought of as a library. When used correctly, generative constructs are fully synthesizable as they are replaced with purely structural code during a phase known as elaboration.

In this work we are concerned only with generative constructs since we are interested in circuits whose structure can be reasoned about. We believe that generative constructs should be used more aggressively. Hardware designers are reluctant to adopt them because current tools do not provide enough support to make their use safe. Despite the fact that generative constructs only makes sense in the context of synthesizable circuits, current tools do not attempt to check for synthesizability statically (i.e. before elaboration), let alone statically check for inconsistencies that Verilog tolerates due to its padding semantics. Most tools accept a description, elaborate it, and then synthesize it into a graph of connected components known as a netlist. If anything goes wrong in this process, an elaboration or synthesis error is thrown depending on when the problem occurred. Even worse: due to Verilog's padding semantics, some violations do not cause errors at all but instead synthesize into a circuit that behaves incorrectly or uses more resources than intended.

To illustrate Verilog's padding semantics and the kind of errors they can hide, let us first consider a module that does not use any generative constructs at all:

```
module invert4(x, y);
   input [3 : 0]  y;
   output [3 : 0]  x;

   assign x = ~ y;
endmodule
```



This module named `invert4` is a 4-bit inverter. It has two 4-pin ports `x` and `y` where `x` is an output port and `y` is an input port. Pins of both ports are indexed

from 0 to 3. The assignment statement uses a bitwise negation operator to invert all input bits and then connect each of them to the corresponding output pin. No padding semantics are used so far but what if the declaration of `y` was changed to `input [4:0] y`? What if instead the declaration of `x` was changed to `output [4:0] x`? In both cases, the assignment statement would be connecting two wires of incompatible widths. Surprisingly, this would still be considered to be a valid description due to Verilog's padding (and trimming) semantics. In the first case the wider input signal will be trimmed discarding the most significant bit, while in the second case the slimmer input signal will be padded with an extra zero as its most significant bit. Note that in the first case, an extra inverter is needed to invert the most significant input bit whose output will be discarded. This is still a concern even if any sensible synthesis tool will discard the extra inverter as well.

Let us consider now a more interesting example: a 4-bit synchronous counter. The counter receives two inputs: a clock `clk` and an enable signal `en`. If `en` is high (equal to 1), then the 4-bit output `count` is incremented at each clock tick. The `next` output can be used to create a wider counter by connecting it to the `en` port of another counter thus cascading both. The schematic diagram in Figure 1 shows how to construct such a counter out of T flip-flops and `and` gates.

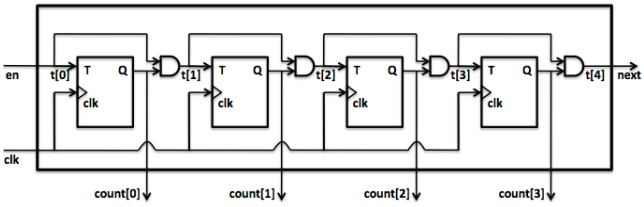


Fig. 1 A 4-bit synchronous counter using T flip-flops

This module can be described using structural Verilog as follows:

```
module  counter4(count, next, en, clk);
   output [3 : 0] count;
   output next;
   input en;
   input clk;
   wire [4 : 0] t;

   assign t [0] = en;
   tflipflop  tff_0 (count [0], t [0], clk);
   assign t [1] = t [0] & count [0];
   tflipflop  tff_1 (count [1], t [1], clk);
   assign t [2] = t [1] & count [1];
   tflipflop  tff_2 (count [2], t [2], clk);
   assign t [3] = t [2] & count [2];
   tflipflop  tff_3 (count [3], t [3], clk);
   assign t [4] = t [3] & count [3];
   assign next = t[4];
endmodule
```

A regularity can easily be detected either by looking at the code or at the schematic diagram. This regularity can be captured to generalize the definition to represent

the whole family of N-bit synchronous counters instead of just the 4-bit synchronous counter in Figure 1. The result is the following parameterized module using a generative loop:

```
module counter_gen(count, next, en, clk);
   parameter N = 4;
   output [N - 1 : 0] count;
   output next;
   input en;
   input clk;
   wire [N : 0] t;
   genvar i;

   assign t [0] = en;
   for(i = 0;i < N;i = i + 1) begin
      tflipflop tff (count [i], t [i], clk);
      assign t [i + 1] = t [i] & count [i];
   end
   assign next = t [N];
endmodule
```

This module named `counter_gen` is parameterized by N whose default value can be anything (in this case we picked 4). This module now acts as a generator that can be run by instantiating the module with a particular parameter value. Each time this module is instantiated as part of a larger design, the width of the counter should be specified, otherwise the default value is used. If this module is instantiated with `N`=4, the elaboration will create a specialized version of it which as expected will be identical to the first version we have written. In all the examples presented in this paper we omit the Verilog keywords `generate` and `endgenerate` that optionally surround generative constructs like the `for`-loop above.

In the generic version an off-by-one error causing array bounds violations can be easily introduced if the hardware designer inadvertently writes the `for`-loop condition as `i<=N`. Again in this case an extra `and` gate and an extra T flip-flop will be created if padding is used naively. Thankfully, current Verilog tools would reject this description, but this will only occur during or after elaboration when the array bounds violations are clearly exposed (In this example, as soon as `tflipflop tff_4 (count [4], t [4], clk)` is generated).

1.1 Importance of Static Consistency Checking

So far we have mentioned two kind of inconsistencies: wire width mismatches and array bounds violations. As we have seen in the previous example, these inconsistencies might not be immediately obvious in designs with generative constructs because the code cannot be directly inspected. A reasonable approach would be to elaborate the code first to eliminate these constructs at which point checking for these inconsistencies is trivial. This is the approach used by current tools, except that they ignore some inconsistencies that are considered legal (and synthesizable) due to padding semantics. The Icarus Verilog compiler for example will ignore matching wire widths violations and ignore array bounds violations unless the index is a constant expression (composed only of constant literals and parameters) in which case it is reported as an elaboration error. We have two strong objections here: first, waiting till elaboration to detect inconsistencies is not a good strategy for the following reasons:

- Elaboration is often time consuming. In such cases, static analysis can save time.
- After elaboration a hardware description can be significantly larger which could make analysis more expensive.
- When an elaboration error is found, it is hard to trace it back to the pre-elaborated design. The designer gets only a cryptic error message referring to generated code that he never wrote.
- Only the specific instantiations of a module which occur in the final circuit are checked. This means that successfully elaborating and synthesizing a circuit does not guarantee that all the modules composing it will never violate array bounds or wire width requirements if instantiated with different parameter values. As an example, consider what would happen if we had accidently forgotten to change the boundaries of `count` from `[3 : 0]` to `[N-1 : 0]` while generalizing the `counter4` module. It is clear that instantiating this module in a larger design with `N` set to any natural number less then or equal to 4 will elaborate successfully but will fail if `N` was set to a value greater than 4. On the other hand, statically checking a generic module before elaboration provides guarantees about all its possible instantiations.

The second objection is with padding semantics hiding inconsistencies. This is even worse than detecting them during or after elaboration for the following reasons:

- Verilog's padding semantics allow designers to write descriptions where wires of different widths can be interconnected. We believe that in most cases, such inconsistencies are the result of typographic errors and not the designer's intent. Going back to the `invert4` example, we believe that replacing the declaration of `y` with `input [4:0] y` should be rejected. If the designer really wants the most significant bit of `y` to be discarded, he can always modify the `assign` statement as follows: `assign x = ~y[3:0]`. Similarly, replacing the declaration of `x` with `output [4:0] x` should be rejected forcing the designer to explicitly change the assign statement to `assign x = ~{1'b0,y}` if he really want `y` to be zero-padded.
- The right value to pad with is often not clear. Should we pad with 0s, 1s, or just extend the most significant bit? Does it make difference if the value we are padding is a 2's complement or an unsigned number?
- Ignored violations will synthesize into potentially "incorrect" circuits. If the intention of the designer is different from the one assumed by the synthesizer, the circuit will occasionally behave incorrectly. However, it will only do so when the computation is affected by the padded/trimmed bits. This kind of error is usually very frustrating and might remain hidden until the circuit gets manufactured.
- Ignored violations might also lead to a circuit using more resources than required.

The most important property of a hardware description is its synthesizability. However we are not interested in descriptions that synthesize into circuits that behave differently from the designer's intentions. Unfortunately, there is no way of capturing such intentions except by requiring the designer to explicitly express them. Verilog's automatic padding semantics, although convenient when they correspond to the designer's intention, are dangerous in all other cases because they tend to hide bugs. This is why we opt to systematically reject systems relying on them. This does not limit what the designer can express because the desired bits can be explicitly appended when this is what is desired. For a description to be "meaningful" and synthesize into a "correct" circuit, wire widths must match and array bounds must be respected.

An ideal tool should be able to statically verify (once and for all) that a module is free from all inconsistencies. If the module is parameterized, the tool's target should be

to verify that this is the case for all possible instantiations (for all acceptable parameter values). In our implementation, it is possible to add some restrictions on the parameter values accepted by such modules using a `where` clause like the one shown in the last two examples of Appendix C. The purpose of a `where` clause is to restrict the possible values of a parameter that can be passed in when instantiating the module. This is a useful feature that makes it possible to explicitly state that a module was not designed to work for all integer values.

1.2 Contributions

Our key contribution is a method to statically verify that a Verilog description is free from 1) Wire width mismatches 2) Array bounds violations. We combine dependent types and constraint generation to reduce the static checking problem to a satisfiability problem. Once reduced, the problem can be handed over to a standard satisfiability modulo theories (SMT) solver. We informally explain our approach by applying it to the `counter_gen` example (Section 2). We also show how the same framework allows us to detect and reject "meaningless" unreachable code (Section 2.5). This ability follows naturally from the constraint gathering required in order to detect other inconsistencies.

A big part of our contribution consists of identifying that Verilog is not only an ideal candidate to be treated and formalized as a statically-typed two-level language (STTL) [3,8,10,11] but is also a perfect example of a dependently-typed language [15, 9] that does not need extra annotations. This observation allows us to formalize our approach by embedding it in a powerful type system. To do so we extend the syntax of Featherweight Verilog (FV) [2] to allow for explicit array width declarations (Section 3.1). FV is a core calculus of structural Verilog with generative constructs that we defined in a previous work. FV is an STTL with an elaboration semantics modeling elaboration and a two-level type system guaranteeing synthesizability [2]. We extend FV's type system to use dependent types and to enforce the required constraints (Section 3.2).

We prove three important properties of our formalization: 1) Type preservation, 2) Type Safety, and 3) Preprocessing soundness (Section 4). Together these three theorems guarantee that a well-typed FV description will always elaborate into an inconsistency-free description with no generative constructs remaining. The complete proofs of these theorems are provided as an appendix.

We provide an implementation (Section 5) of these ideas in the form of a Verilog Pre-Processor that we call VPP. VPP performs elaboration on Verilog descriptions that it can prove to be well-typed and rejects all other descriptions. The output is a structural Verilog description free from any generative constructs. Finally, we discuss the limitations of our approach (Section 5.1) and evaluate the performance of our implementation to demonstrate its applicability and scalability (Section 5.2).

2 Approach

Dependent types, recording control flow information, and generating consistency constraints are three key ingredients to reduce the problem of static consistency checking to a satisfiability problem. In the following subsections, we informally describe each of these and how everything is put together.

2.1 Dependent Types

A dependent type is a type that depends on a value. For our purposes, we use dependent types to enrich wire types with their upper and lower bounds. This follows naturally from the way arrays are declared in Verilog. For example if `x` is declared using the following statement: `input [N:M] x` then `x` has type `wire(min(N,M),max(N,M))` where `min(N,M)` is its lower bound and `max(N,M)` is its upper bound, instead of having the simpler but less informative type `wire array`.

Using dependent types, the type of `count` in module `counter_gen` is `wire(min(N-1, 0),max(N-1,0))`. Similarly the type of `t` in the same module is `wire(min(N,0),max(N, 0))`. Types of other input/output signals is just `wire` since they are all single bit ports.

2.2 Recording Control Flow Information

The information recorded in the type is not quite all the information we need, nor is it all the information we can gather. We can gather additional information from generative constructs (both `if`-conditions and `for`-loops). In case of loops, the additional information is the loop invariant whose inference is undecidable in the general case but very easy to infer if the loop construct is restricted enough. In this work we restrict our attention to simple loops that can be easily analyzed. Our prototype implementation rejects descriptions containing loops whose invariants cannot be inferred.

In the `counter_gen` example we do not have a conditional but we do have a generative loop. We can easily infer that within the body of the loop, the loop index i is always less than N and always greater than or equal to 0.

2.3 Generating Consistency Constraints

Given the information we know about wire arrays and the information we have collected by analyzing the control flow, we now need to prove that inconsistencies never occur in a given description. To do so we generate a constraint that we need to verify for each potential inconsistency source. In particular we generate constraints whenever we encounter any of these:

- Wire Assignment: width of the left hand side (LHS) must be equal to the width of the right hand side (RHS)
- Module Instantiation: passed wires must have widths compatible with those in the module signature
- Array Access: array indices are within bounds

As an example, let us take the T flip-flop instantiation statement in the body of the loop in module `counter_gen`. This statement generates 7 constraints (some of which are redundant). First we need to make sure the passed wires have widths compatible with the signature of the T flip-flop module. The T flip-flop module outputs a single bit, and requires a single bit clock and another single bit input value. Since the widths of all the passed signals is actually 1, the required constraint (generated 3 times) is $1 = 1$ and is trivially true. Second we need to make sure that `count[i]` does not cause any array bounds violation which requires 2 constraints: $i \geq min(N-1, 0)$ and $i \leq max(N-1, 0)$ where $min(N-1, 0)$ and $max(N-1, 0)$ are the lower and upper

bounds of `count` as recorded in its type. Similarly the constraints $i \geq min(N, 0)$ and $i \leq max(N, 0)$ are generated to make sure that `t[i]` does not cause array bounds violations.

2.4 Verifying Consistency

We need to prove that each constraint holds given the type information and the collected information at this point. In general we find ourselves required to prove that the conjunction of a set of facts (givens) implies a different fact (consistency constraint) for all possible variable values. A bit more formally we need to prove the correctness of a logic formula of the form: $\forall x_1, x_2, ..., x_n . \bigwedge_{i=1}^{k} c_i \Rightarrow c$ where c_i represent known facts and c is the consistency constraint. In the following sections this same formula will be written more concisely as $\langle c_i \rangle \rhd c$

Verifying the truth of this universally quantified formula can be converted into a satisfiability problem by negating the formula and checking for its unsatisfiability. Negating the above formula and using basic logic rules we obtain a formula of the form: $\exists x_1, x_2, ..., x_n . \bigwedge_{i=1}^{k+1} c_i$ where $c_{k+1} = \neg c$. Once converted to a satisfiability problem, it can can be handed over to a standard SMT solver.

For example, let us take the $i \leq max(N, 0)$ constraint generated by the instantiation statement in `counter_gen`. We basically need to prove that this always holds given what we have inferred about i. So we need to prove that: $\forall i, N.(i < N \wedge i \geq 0) \Rightarrow i \leq max(N, 0)$. We can prove this by proving the unsatisfiability of its negation: $\exists i, N.(i < N \wedge i \geq 0 \wedge i > max(N, 0))$. Otherwise stated we want to make sure that no integer value i can simultaneously satisfy the information we inferred from the loop invariant and be greater than the upper bound of `count`. If such a value is found then our program should be rejected.

2.5 Unreachable Code Detection

The same satisfiability framework can be used to serve a slightly different purpose. It can be used to detect unreachable code. Consider the following example:

```
module adder (sum, a, b);
   parameter N = 8;
   input [N-1 : 0] a;
   input [N-1 : 0] b;
   output [N : 0] sum;
   if(N<16)
      if(N<8)
         ripple_adder #(N) radder (sum, a, b);
      else
         cla_adder #(N) cladder (sum, a, b);
   else
      cselect_adder #(N) csadder (sum, a, b);
endmodule
```

This is an N-bit adder that will add `a`, `b` and produce their sum. Internally this adder is either a ripple, a carry look ahead, or a carry select adder depending on the parameter value used to instantiate it. Note that the value 8 assigned to `N` is only a default value, any different value can be used when the module is actually instantiated as part of a bigger design. What if the condition `N<16` is replaced by `N>16`? The module

still means something except that `ripple_adder #(N) radder (sum, a, b)` will be unreachable as `N` cannot be greater than 16 and less then 8 at the same time. If such code is presented it usually indicates a bug and should be pointed out or even better it should be rejected as it is "meaningless". Clearly this can be done using the same framework we just described. All we need to do is make sure that all the facts collected by analyzing loops and conditionals are consistent with each other.

3 Featherweight Verilog

To be able to prove that our approach is reasonable, we need to formalize it and prove that it can indeed be used to statically check that a description is free from the aforementioned inconsistencies. We basically want to define a type system that will only consider inconsistency-free descriptions to be well typed. In a previous work [2] we defined FV, a core calculus of structural Verilog with generative constructs. We opted for defining FV as a statically-typed two-level language to be able to naturally model the elaboration phase, which was defined using big-step operational semantics. In [2] we proved three key properties of FV:

- Type Preservation: the resulting description obtained from elaborating a well-typed description is well-typed
- Type Safety: preprocessing a well-typed description will never fail
- Preprocessing Soundness: after elaboration a description is guaranteed to be free from any generative constructs

Combining these results and by defining synthesizability we were able to show that a well-typed description was guaranteed to be synthesizable.

In this paper we extend FV definitions to allow us to statically check for inconsistencies. In the following subsections we present the required extensions to the FV syntax, operational semantics and type system. To make these extensions easier to spot and focus on, we highlight them in gray. As expected the most interesting changes occur in the type system. We also prove that the key properties of FV still hold.

3.1 FV Syntax

In this section we reprise the FV grammar definition from [2] with minor modifications. 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:

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

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 four sequences: (1) module parameter names $\langle x_i \rangle^{i \in I}$, (2) port declarations (carrying direction, type, and name for each port) $\langle d_j\ t_j\ s_j \rangle^{j \in J}$, (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`, `int`, or an array with upper and lower bounds. 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, an integer v, or an operator application $f \langle e_i \rangle^{i \in I}$. Conditional statements and `for`-loops are FV's main generative constructs. In FV's syntax generative constructs are not surrounded by the `generate` and `endgenerate` keywords.

The main difference from the syntax defined in [2] is that wire declarations have explicit upper and lower bounds associated with them unless they are single bit wires. This in turn requires both module ports and local variables to have types associated with them. Making these explicit makes FV closer to the original Verilog syntax. We had omitted those previously because we were not concerned by them for synthesizability issues. It is important to note that in FV syntax (just like in Verilog syntax) the expressions used to specify the array bounds can be in any order. This is different from the convention we use in our type system that always assumes that the first expression is the lower bound expression and that the second one corresponds to the upper bound.

The second difference is that `for`-loop headers are syntactically restricted in a way that makes loop invariants easy to infer. Finally, integer values are not restricted to 32 bit values. An integer is now a non-empty sequence of 0s and 1s of arbitrary length to be able to assign them to wires of various width while being able to verify the consistency of such assignments.

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

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

For detailed comparison between FV syntax and the syntax of the corresponding Verilog subset, we refer the reader to [2].

3.1.1 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 $\langle X_i \rangle$.

3.2 FV Type System

As previously defined, the used type system is a two-level type system. In the terminology of two-level languages, preprocessing 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. $\vdash^n$.

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

$$
\begin{array}{lrl}
\textit{Module Type} & M & ::= \langle x_i \rangle^{i \in I}\ \langle d_j\ t_j \rangle^{j \in J} \\
\textit{Operator Signatures} & \Sigma & \in \Pi i.\, \mathbb{O} \times \mathbb{N} \times \mathbb{T}^i \to \mathbb{T} \\
 & & \\
\textit{Level} & n & ::= 0 \mid 1 \\
\textit{Module Environment} & \Delta & ::= [\,] \mid m : M :: \Delta \\
\textit{Variable Environment} & \Gamma & ::= [\,] \mid s : d\ t :: \Gamma \mid x : d\ t :: \Gamma \\
\textit{Level 1 Variable Env.} & \Gamma^+ & ::= [\,] \mid s : d\ t :: \Gamma^+ \\
 & & \\
\textit{Constraint Environment} & C & ::= [\,] \mid c :: C \\
\textit{Constraint} & c & ::= e\, R\, e \mid \neg c \\
\textit{Relational Operator} & R & ::= < \mid > \mid \leq \mid \geq \mid = \mid \neq
\end{array}
$$

A module type now consists of a sequence of parameters and a sequence of directions and types for ports. These types might be dependent on the parameters values. The rest is unchanged: 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. The operator signature is itself a dependent type since it depends on

the number of operands i. This dependence is indicated by the usage of Π. 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.

We also added a constraint environment C which is a list of constraints c known to be true. A constraint can be a relation between two Verilog expressions, or a negation of another constraint.

To make the typing rules more readable, we recursively define a function *width* that returns the width given type information:

$$\begin{aligned} width(int) &= \infty \\ width(wire) &= 1 \\ width(t(e_1, e_2)) &= (e_2 - e_1 + 1) * width(t) \end{aligned}$$

Note that width of `int` is considered to be infinite (or unbounded). This means that integer variables, such as parameters can be of arbitrary sizes and therefore can never be assigned to wires. The width of a `wire` is 1 and the width of an array of elements of width w is equal to w multiplied by the array size, which is the upper bound − lower bound + 1. Note that at this point we assume that the boundaries are ordered. This is achieved using a function *order* that returns a type where bounds are ordered given a type with no restriction on the order of bounds. Here is the recursive definition of *order*:

$$\begin{aligned} order(int) &= int \\ order(wire) &= wire \\ order(t(e_1, e_2)) &= order(t)(min(e_1, e_2), max(e_1, e_2)) \end{aligned}$$

For types that do not have bounds, *order* is just an identity function otherwise *order* makes sure that the lower bound (which the minimum of the given bounds) is presented before the lower bound (which is the maximum of the given bounds).

Figures 2 and 3 define the rules for the judgment $\vdash p$. A circuit description p is well-typed when this judgment is derivable. These rules are the same as presented in [2] except for the highlighted parts. These modifications were done for three main reasons: 1) The type system now requires dependent types. 2) It is also required to maintain an additional environment C of given constraints. 3) Typing rules now have additional premises to guarantee consistency.

For a detailed explanation of each typing rule, we refer the reader to [2]. The modification in T-Prog is due to the module signature change. In addition to the trivial changes, T-Body now requires the types of ports and local variables to be well-typed themselves in an environment that only contains parameters. The typing judgment for types is added in the end of Figure 3. T-Mod now requires the widths of signals passed to modules to be compatible with the instantiated module signature after substitution. T-Assign1 and T-Assign2 now check for wire width mismatches. T-If appends the known constraints environment depending on the branch. Similarly T-For appends the constraint environment with the loop environment. To make sure the loop terminates, e_3 is required to be greater than 0 and e_2 is not allowed to use the loop index y. T-Idx and T-Rg add the necessary conditions to check for array bounds violations.

This type system could also be used to detect unreachable code although no explicit extensions need to be added. All we need to do is to change C's definition to be:

Satisfiable Constraint Environment $C ::= [\,] \mid c :: C$

A satisfiable constraint environment cannot contain conflicting constraints.

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

$$\frac{\begin{array}{l}\{\Gamma \; ; C \;\vdash^0 e_i : \{\texttt{in}\}\ \texttt{int}\} \qquad \Delta(m) = \langle x_i \rangle\ \langle d_j \;\; t_j \rangle \\ \{\Gamma \; ; C \;\vdash^1 l_j : d'_j \; t'_j\} \qquad \{d_j \subseteq d'_j\} \\ \{C \rhd width(t_j\{[x_i \mapsto e_i]\}) = width(t'_j)\}\end{array}}{\Delta; \Gamma \; ; C \;\vdash m\langle e_i \rangle\ \langle l_j \rangle}\ (\mathrm{T-Mod})$$

$$\frac{\begin{array}{l}\texttt{out} \in d_1 \qquad \texttt{in} \in d_2 \\ \Gamma \; ; C \;\vdash^1 l : d_1 \; t_1 \\ C \rhd width(t_1) = i\end{array}}{\Delta; \Gamma \; ; C \;\vdash \texttt{assign}\ l\ (0 \mid 1)^i}\ (\mathrm{T-Assign1}) \qquad \frac{\begin{array}{l}e \neq (0|1)^+ \\ \texttt{out} \in d_1 \qquad \texttt{in} \in d_2 \\ \Gamma \; ; C \;\vdash^1 l : d_1 \; t_1 \\ \Gamma \; ; C \;\vdash^1 e : d_2 \; t_2 \\ C \rhd width(t_1) = width(t_2)\end{array}}{\Delta; \Gamma \; ; C \;\vdash \texttt{assign}\ l\ e}\ (\mathrm{T-Assign2})$$

$$\frac{\begin{array}{l}\Gamma \; ; C \;\vdash^0 e : \{\texttt{in}\}\ \texttt{int} \\ \{\Delta; \Gamma \; ; C, e \;\vdash P_i\} \qquad \{\Delta; \Gamma \; ; C, \neg e \;\vdash P_j\}\end{array}}{\Delta; \Gamma \; ; C \;\vdash \texttt{if}\ e\ \texttt{then}\langle P_i \rangle \texttt{else}\langle P_j \rangle}\ (\mathrm{T-If})$$

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

Fig. 2 Type System For Parallel Statements

3.3 FV Operational Semantics

In [2] we formalized the elaboration process by defining 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.

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

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

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 $X_\perp$.

The required notion of substitution is extended to define substitution inside types, type environments, and constraint environments. Most of the operational semantics remain unchanged except for minor adjustments to accommodate for the syntax change. The only notable change is in the module instantiation rule (E-Mod) where the types of

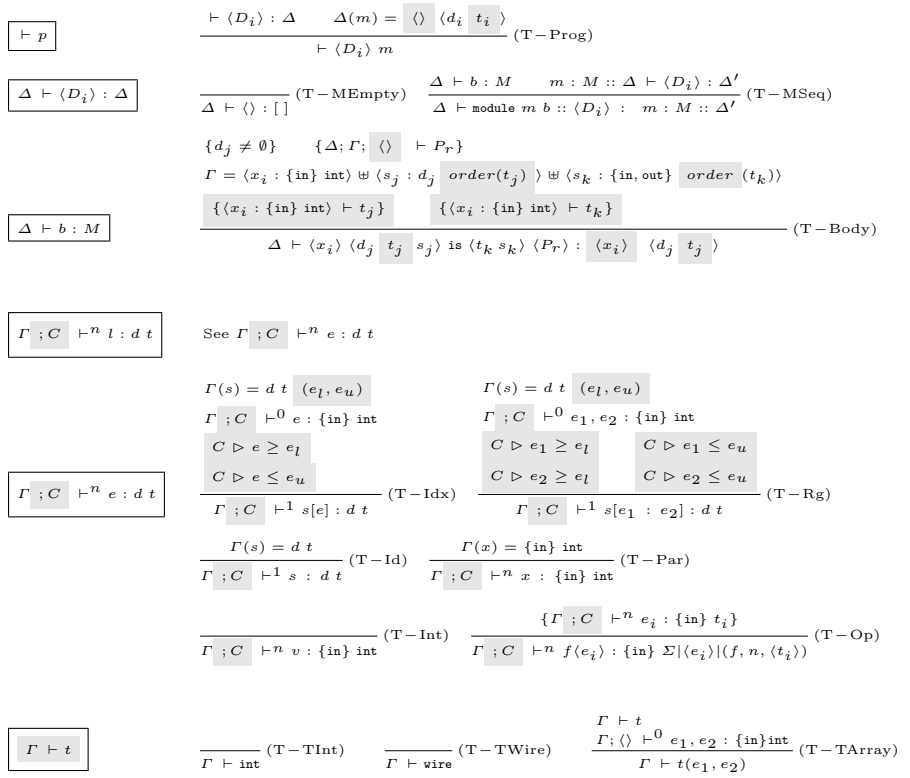


Fig. 3 Type System For All Other Constructs

the newly created module need to be updated using substitution to reflect the values of the parameters that were used in the instantiation. We display the substitution definition with the required extensions in Figure 4 and the definition of the operational semantics in Figure 5. The potential errors of operational semantics errors are defined in Appendix A.

4 Technical Results

In this section we restate the main theorems from our previous work and show that they still hold. The complete proofs of these theorems are in Appendix B.

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

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

Proof (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:

$P[x \mapsto v]$

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

$l[x \mapsto v]$ See $e[x \mapsto v]$

$e[x \mapsto v]$

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

$c[x \mapsto v]$

$$
\begin{aligned}
(e_1\ R\ e_2)[x \mapsto v] &= e_1[x \mapsto v]\ R\ e_2[x \mapsto v] \\
(\neg e)[x \mapsto v] &= \neg(e[x \mapsto v])
\end{aligned}
$$

$\Gamma[x \mapsto v]$

$$
\begin{aligned}
[\,][x \mapsto v] &= [\,] \\
(s : d\ t :: \Gamma)[x \mapsto v] &= s : d\ t[x \mapsto v] :: \Gamma[x \mapsto v] \\
(x : d\ t :: \Gamma)[x \mapsto v] &= x : d\ t[x \mapsto v] :: \Gamma[x \mapsto v]
\end{aligned}
$$

$t[x \mapsto v]$

$$
\begin{aligned}
\texttt{int}[x \mapsto v] &= \texttt{int} \\
\texttt{wire}[x \mapsto v] &= \texttt{wire} \\
t(e_1, e_2)[x \mapsto v] &= t[x \mapsto v](e_1[x \mapsto v], e_2[x \mapsto v])
\end{aligned}
$$

Fig. 4 Substitution

Theorem 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.

We also want to prove type safety:

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

Proof Proving Type Safety Theorem (Theorem 2) is straightforward. Since $\vdash p$ and $p \overset{1}{\hookrightarrow} p'$, we know from Theorem 1 that $\vdash p'$ and since we do not have any typing rules in our type system that would consider `err` to be well-typed then we can directly conclude that $p' \neq \texttt{err}$

Finally we want to prove preprocessing soundness, which means that preprocessing produces fully expanded terms. The set of fully expanded terms is defined as follows:

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

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

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

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

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

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

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

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

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

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

$$\frac{e_1 \overset{0}{\hookrightarrow} v_1 \qquad e_2 \overset{0}{\hookrightarrow} v_2 \qquad v_1 \geq v_2}{\langle D_i \rangle \vdash \texttt{for}(y = e_1;\ y < e_2;\ y = y + e_3)\ \langle P_k \rangle \overset{1}{\hookrightarrow} \langle \rangle, \langle \rangle}\ (\text{E}-\text{ForFalse})$$

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

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

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

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

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

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

Fig. 5 Operational Semantics

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

Theorem 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.

5 Implementation

A prototype implementation of the Verilog Pre-Processor (VPP) is available for download at `http://www.resource-aware.org/do/view/RAP/VPP`.

VPP includes a type checker based on the typing rules defined in Section 3.2. If the given description is well-typed then it is elaborated into an equivalent description that is guaranteed to be inconsistency free.

To check for satisfiability conditions, VPP makes use of Yices, a state of the art SMT solver [1]. Since VPP is developed in OCaml, we developed an OCaml library to communicate with the C APIs provided by Yices. Yices is needed in 2 different situations:

1. When gathering constraints from conditionals and loops: each new piece of information must not conflict with previously collected information. Each time the set of collected constraints is extended, VPP passes the extended set to Yices to verify its satisfiability. In case it is unsatisfiable, the construct causing the extension is considered ill-typed as it guards an unreachable block of code.
2. When verifying consistency requirements: whenever a new consistency requirement is reached, VPP passes its negation along with the set of collected constraints to Yices looking for unsatisfiability. In case the extended set is satisfiable, the construct being type-checked is considered ill-typed as it violates one of the consistency constraints.

Figure 6 illustrates the interaction between VPP and Yices. Given an unelaborated description, VPP does static type checking and the major part of that is generating a collection of integer satisfiability problems that are handed over to Yices. VPP finally decides the well-typedness of the description based on its own typing rules and the set of Yices responses.

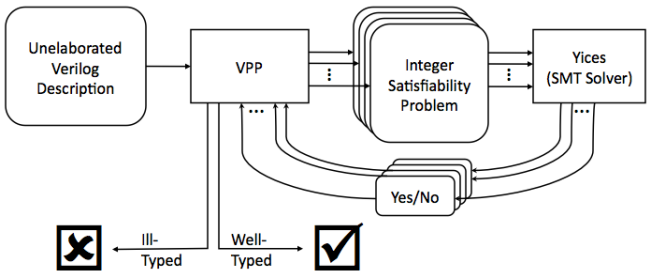


Fig. 6 VPP Interaction with Yices

In particular VPP uses the integer linear arithmetic and the uninterpreted function theories of Yices. The integer linear arithmetic theory is used in various places, especially when translating Verilog expressions into Yices expressions. The uninterpreted functions have been very useful to define nonlinear functions needed to handle the limitations of our approach as explained in the following section.

5.1 Limitations

The limitations of our implementation and of our approach in general are the same as the limitations of the SMT solver used. In general all the constraints need to be linear relations for any SMT solver (like Yices) to be able to check their satisfiability. However, this limitation turned out to be less problematic for two reasons:

- In our experience most constraints that come up in structural hardware descriptions tend to be linear unless these descriptions are written in an intentionally obscure style. An example would be the usage of a single dimension array to represent what would normally be a two-dimensional array of wires. In this case the upper bound of the single dimension array is likely to be the product of two parameters immediately generating a nonlinearity. Rewriting this description to use a two-dimensional array gets rid of the problem and makes the code more readable.
- In the cases where the nonlinearity cannot be avoided or results from the natural way of describing a circuit, uninterpreted functions can be used to convert the nonlinearity constraint back to a linear one as explained below.

The nonlinear function *min* can be defined in Yices as follows:

```
(define min:: (-> x::int y::int
                  (subtype(r::int)
                          (and (or (= r x)(= r y))
                               (<= r x)
                               (<= r y)))))
```

We define *min* as an uninterpreted function that takes two integers `x` and `y` and returns an integer `r` such that its value is either equal to `x` or `y` while being at the same time less than or equal to both. Of course, the function *max* can be defined in a similar fashion.

Not all nonlinearities can be handled using uninterpreted functions as nicely as *min* and *max* were handled. Another nonlinear operation that we used more than once in structural descriptions is raising the number 2 to an integer power. Here is how we can define *pow*2 in Yices:

```
(define pow2::(-> nat nat))
```

We define *pow*2 as an uninterpreted function that takes a natural number and returns a natural number. As opposed to *min* and *max* definitions, the *pow*2 definition is a rather incomplete definition. To help alleviate this issue, we can complement the definition with a few constraints of the form:

```
(assert (= (pow2 1) 2))
(assert (= (pow2 2) 4))
(assert (= (pow2 3) 8))
```

Currently these constraints are added manually when needed depending on the description being analyzed to help Yices prove the correctness of a particular definition, but it is not hard to conceive a program that would assert the power constraint that it needs automatically in certain cases.

Similarly, non-constant multiplication can be defined in Yices as an uninterpreted function as follows:

```
(define mult::(-> int int int))
```

Table 1 VPP's Performance

Circuit	Code Size (lines)	P (msec)	TC (msec)	PP (msec)	SAT (msec)
Inverter	10	0.32	0.13	0.05	18
Counter	20	0.41	0.23	0.05	33
Parity	13	0.34	0.18	0.06	28
Ripple Adder	27	0.51	0.29	0.08	41
Carry Select	59	0.77	0.49	0.26	64
Decoder	22	0.40	0.32	0.12	64
Multiplexer	21	0.46	0.45	0.07	32
Multiplier	39	0.66	0.52	0.16	126

Clearly, the support provided by uninterpreted functions in general is limited and it is not enough to prove the type safety of an arbitrary description. We assume that any module that our type system cannot prove safe is not well-typed. This of course means that there exists perfectly correct modules that will be rejected by our conservative type system.

5.2 Performance Evaluation

We have argued about the advantages and the feasibility of static checking before elaboration; however, static checking adds some overhead. In this section we empirically evaluate such overhead. In particular, we show that the computational overhead of the SMT solver does not exceed the benefits of static checking.

We wrote some descriptions of circuit families and ran them through VPP to evaluate its performance. Our benchmark is composed of the following generic circuits: 1) an N-bit inverter (presented earlier), 2) a synchronous N-bit counter (presented earlier), 3) an N-bit linear parity generator using 2-input `xor` gates only, 4) an N-bit ripple adder using full adders, 5) an N-bit carry-select adder block, 6) an N-by-2^N decoder, 7) a 2^N-by-1 multiplexer, and 8) an N-by-M purely combinational multiplier. The source code of these examples can be found in Appendix C.

VPP's type checker is based on the type system defined in Section 3.2. As such it is a modular checker and we therefore expect it to be reasonably efficient. Table 1 shows the time required to parse (P), type check (TC), and preprocess (PP) the Verilog examples mentioned above along with the code size for each example. The type checking time does not include the time spent by Yices to solve satisfiability problems. This time is shown separately on the last column (SAT).

To accurately time these examples, we measured the timings of running each example one thousand times and divided the obtained values by one thousand. These experiments were conducted on a 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. Yices consumes the most amount of time but it is still in the order of milliseconds. The multiplier example in particular requires 126 ms solving satisfiability problems despite not being the longest description because of its extensive use of two-dimensional arrays. In general we did not encounter any performance problems with the examples we studied so far.

6 Related Work

To the best of our knowledge, no one has considered static checking of Verilog descriptions before. Current tools do perform some checks but these are mostly done after elaboration. On the other hand, various techniques for static array bounds checking for general purpose languages have been explored earlier. Static buffer overflow analysis for C-like programming languages has been extensively studied. Checking legacy code is usually very expensive as it requires rather complex inter-procedural analysis. Examples of non-modular checkers include Polyspace Ada/C/C++ Verifier [12], C Global Surveyor [13]. This approach is of limited scalability. An alternative approach would be to explicitly specify the preconditions and postconditions of each function using special annotations. When this information is available, the checker only needs to check each function independently, making the task of detecting violations much more precise and scalable. Many annotation-based approaches have been proposed in the literature. Splint [7] is such an example. It does lightweight analysis but it is neither sound nor complete. ESPX [4] is sound and complete and uses an incremental annotation approach. If the code is fully annotated, it is comprehensively checked for buffer overflows. ESPX also uses SALInfer, an annotation inference engine that facilitates the annotation process by automating some of it. Annotation based approaches are not really useful until the programmer has manually annotated the source code, which requires him to learn the annotation language and spend a significant amount of time actually adding the annotations.

The reason behind that our type checker does not need to do inter-modular analysis and it does not require the programmer to put any additional annotations is that each module has a well defined signature that specifies its inputs, outputs and the boundaries of each of these. This readily available information provides us with the needed pre- and post-conditions. Also we do not want to rely on how a module is instantiated to decide its well-typedness. According to our definitions, a module's well-typedness is independent from how it is used and that is why a well-typed parameterized circuit description can safely be used in any design without further checking.

Our approach was inspired by Xi and Pfenning's approach in DML [14], which is a dependently typed extension of ML. The main differences that distinguish our work are:

- DML uses type inference requiring two passes over the code: the first to infer types and the second to generate constraints based on dependent types annotations explicitly added by the programmer. FV does not require the type inference pass or any annotations as all the types and their associated widths are explicitly declared.
- FV is formalized as a two-level language, while DML is not. Some of the challenges we encountered in the formalization process were due to the two-level nature of FV.
- The consistency checks needed for Verilog are different and emerge from understanding hardware constraints.
- DML uses Fourier Variable Elimination for constraint solving while we use a more general SMT solver, which gives us more power because of the availability of other theories.

7 Conclusions and Future Work

In this paper we have shown how dependent types can be combined with constraint generation to create a type system that is powerful enough to statically detect array bounds violations, wire assignment inconsistencies, and unreachable code. We have proved type preservation and safety of our type system with respect to elaboration and proved the soundness of elaboration itself. Nowadays it is still the case that hardware designers find themselves making heavy use of Perl scripts or emacs advanced editing modes to help them generate hardware modules of various sizes. These techniques are hard to read, reason about, and verify. VPP (or any other tool implementing the main ideas in our paper) allows designers to safely describe circuit families using structural and generative constructs.

An important goal of this project is to increase the productivity of hardware designers by giving them the means to describe circuits at a higher level of abstraction while still giving them enough control over the generated hardware. As such we would like to take a closer look at examples that are hard to express using the currently available generative constructs and add constructs that enable us to describe them in a convenient manner. These constructs will be expanded away by VPP to make sure the final outcome is in standard Verilog to allow integration with current tool chains.

We also believe that dependent types can be used to capture physical properties of the hardware design. In particular, we would like to be able to use the current framework to estimate the number of gates, the area, and the power required by a certain description.

References

1. Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for DPLL(T). *Computer Aided Verification*, 4144:81–94, 2006.
2. Jennifer Gillenwater, Gregory Malecha, Cherif Salama, Angela Yun Zhu, Walid Taha, Jim Grundy, and John O'Leary. Synthesizable high level hardware descriptions: using statically typed two-level languages to guarantee Verilog synthesizability. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 41–50, New York, NY, USA, 2008. ACM.
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. Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 232–241, New York, NY, USA, 2006. ACM.
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 Verilog Hardware Description Language*. Number 1364-2005 in IEEE Standards. IEEE, 2005.
7. David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.
8. Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, 1988.
9. Benjamin C. Pierce, editor. *Advanced Topics in Types and programming languages*. MIT Press, Cambridge, MA, USA, 2005.
10. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
11. 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.

12. PolySpace Technologies. http://www.polyspace.com, 1999.
13. Arnaud Venet and Guillaume P. Brat. Precise and efficient static array bound checking for large embedded C programs. In William Pugh and Craig Chambers, editors, *PLDI*, pages 231–242. ACM, 2004.
14. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.
15. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM.

A Preprocessing Errors

Figures 7 and 8 explicitly formalize the errors that can occur during preprocessing. These are unchanged from [2] except for minor syntactic adjustments.

B Proofs and Auxiliary Results

B.1 Proof of the Substitution Lemma (Lemma 1)

We first need to prove that if a constraint is provable given an constraint environment, substituting all the occurrences of variable inside that constraint with a value that is consistent with the environment do not affect its provability.

Lemma 2 (Substitution for constraints) *if $C \rhd c$ and $C, x = v$ is satisfiable then $C \rhd c[x \mapsto v]$*

Proof If c is provable given C. This means that c is true for all possible values of x that do not violate the constraints in C. Therefore if all occurrences of x in c are replaced by v such as $x = v$ is still consistent with C then the new constraint $c[x \mapsto v]$ is still true given C. This means that $C \rhd c[x \mapsto v]$ still holds.

To prove Lemma 1, we need to establish the same property for smaller constructs first. We first state and prove Lemmas 3 and 4 that establish that substitution preserves typing for e and l respectively.

Lemma 3 (Substitution for expressions) *If $\Gamma, x : d_1\ t_1; C \vdash^{n_2} e : d_2\ t_2$ and $\Gamma; C \vdash^{n_1} v : d_1\ t_1$ and $C, x = v$ is satisfiable then $\Gamma[x \mapsto v]; C \vdash^{n_2} e[x \mapsto v] : d_2\ t_2[x \mapsto v]]$*

Proof We proceed by induction on the typing derivation of e. We have six distinct cases:

- **Case Id:** e is s and from the substitution rule $e[x \mapsto v]$ is s. From T-Id, we get that $n_2 = 1$ and that $(\Gamma, x : d_1\ t_1)(s) = d_2\ t_2$ and since s cannot be equal to x than $\Gamma(s) = d_2\ t_2$. Therefore $(\Gamma[x \mapsto v])(s) = d_2\ t_2[x \mapsto v]$. Using T-Id again we get $\Gamma; C \vdash^{n_2} s : d_2\ t_2[x \mapsto v]$.
- **Case Index:** e is $s[e_1]$ and from the substitution rule $e[x \mapsto v]$ is $s[e_1[x \mapsto v]]$. From T-Idx, we know that $n_2 = 1$, that $(\Gamma, x : d_1\ t_1)(s) = d_2\ t_2(e_l, e_u)$, that $\Gamma, x : d_1\ t_1; C \vdash^0 e_1 : \{\texttt{in}\}\ \texttt{int}$, that $C \rhd e_1 \geq e_l$ and that $C \rhd e_1 \leq e_u$. Since e_1 has a smaller typing derivation tree than e, we can apply the induction hypothesis

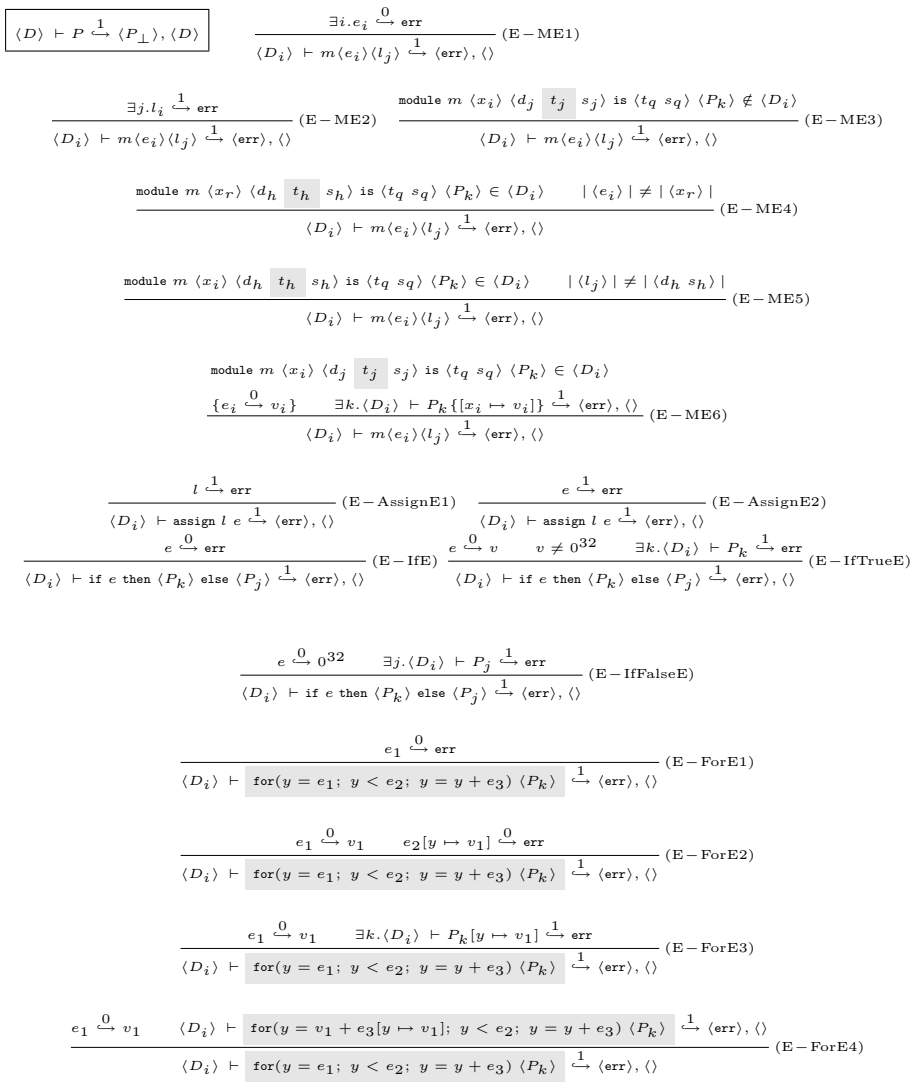


Fig. 7 Operational Semantics Errors For Parallel Statements

to get that $\Gamma[x \mapsto v]; C \vdash^0 e_1[x \mapsto v] : \{\texttt{in}\}\ \texttt{int}$. and again since s cannot be equal to x than $\Gamma(s) = d_2\ t_2(e_l, e_u)$. This also means that $(\Gamma[x \mapsto v])(s) = d_2\ t_2[x \mapsto v](e_l[x \mapsto v], e_u[x \mapsto v])$ Applying Lemma 2 and by using T-Idx one more time we get that $\Gamma; C \vdash^{n_2} s[e_1[x \mapsto v]] : d_2\ t_2[x \mapsto v]$.

- **Case Range:** e is $s[e_1 : e_2]$ and from the substitution rule $e[x \mapsto v]$ is $s[e_1[x \mapsto v] : e_2[x \mapsto v]]$. Following the same steps of the previous case but using T-Rg instead of T-Idx and applying the induction hypothesis twice we get the desired result.
- **Case Param:** There are two sub-cases to consider:

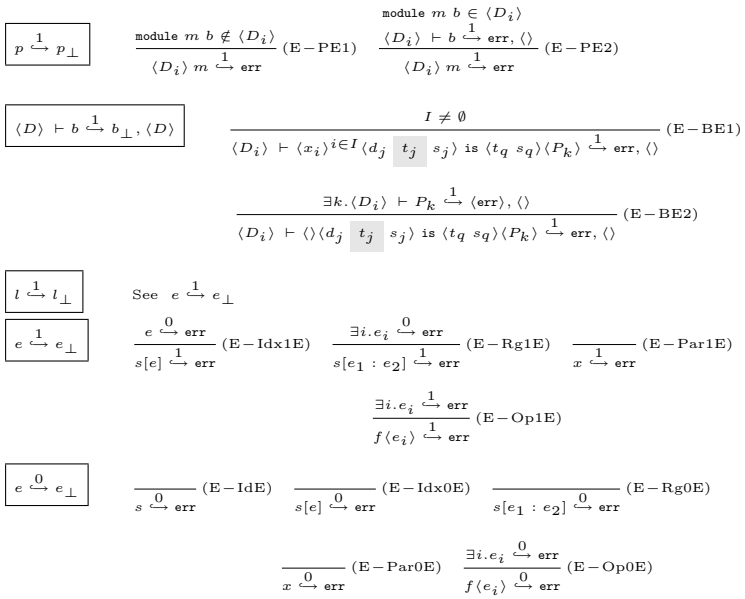


Fig. 8 Operational Semantics Errors For All Other Constructs

  - e is x: In this case $e[x \mapsto v]$ is v from the substitution rule. From T-Par we get that $d_2\ t_2 = \{\texttt{in}\}\ \texttt{int}$. But from T-Int we know that $\Gamma; C \vdash^{n_2} v : \{\texttt{in}\}\ \texttt{int}$. And we know that $\texttt{int}[x \mapsto v] = \texttt{int}$. Therefore $\Gamma[x \mapsto v]; C \vdash^{n_2} v : d_2\ t_2[x \mapsto v]$
  - e is y where $y \neq x$: In this case $e[x \mapsto v]$ is y from the substitution rule. From T-Par we get that $(\Gamma, x : d_1\ t_1)(y) = \{\texttt{in}\}\ \texttt{int}$. But since $y \neq x$ then $\Gamma(y) = \{\texttt{in}\}\ \texttt{int}$. Using T-Par again, we get that $\Gamma[x \mapsto v]; C \vdash^{n_2} y : d_2\ t_2[x \mapsto v]$
- **Case Int:** e is v' and from the substitution rule $e[x \mapsto v]$ is v'. From T-Int the result is immediate because the type of v' does not depend on Γ.
- **Case Op:** e is $f\langle e_i \rangle$ and $e[x \mapsto v]$ is $f\langle e_i[x \mapsto v] \rangle$ from the substitution rule. From T-Op, we know that $d_2\ t_2 = \{\texttt{in}\}\ \Sigma | \langle e_i \rangle | (f, n_2, \langle t_i \rangle)$ and that $\{\Gamma, x : d_1\ t_1; C \vdash^{n_2} e_i : \{\texttt{in}\}\ t_i\}$. Since each e_i has a smaller typing derivation than e, we can use the induction hypothesis to get $\{\Gamma[x \mapsto v]; C \vdash^{n_2} e_i[x \mapsto v] : \{\texttt{in}\}\ t_i[x \mapsto v]\}$. Therefore by using T-Op again, we get that $\Gamma[x \mapsto v]; C \vdash^{n_2} f\langle e_i[x \mapsto v] \rangle : \{\texttt{in}\}\ \Sigma | \langle e_i[x \mapsto v] \rangle | (f, n_2, \langle t_i[x \mapsto v] \rangle)$. Noting that $| \langle e_i[x \mapsto v] \rangle | = | \langle e_i \rangle |$ we reach the desired conclusion.

Lemma 4 (Substitution for LHS values) *If* $\Gamma, x : d_1\ t_1; C \vdash^{n_2} l : d_2\ t_2$ *and* $\Gamma; C \vdash^{n_1} v : d_1\ t_1$ *and* $C, x = v$ *is satisfiable then* $\Gamma[x \mapsto v]; C \vdash^{n_2} l[x \mapsto v] : d_2\ t_2[x \mapsto v]$

Proof This follows directly from Lemma 3 because any l is also an e.

Now we have all the auxiliary results needed to prove the substitution lemma (Lemma 1). The proof works as follows:

Proof We proceed by case analysis on the typing derivation. We consider four cases:

- **Case Module Instantiation:** P is $m\,\langle e_i\rangle\langle l_j\rangle$ and from the substitution rule $P[x \mapsto v]$ is $m\,\langle e_i[x \mapsto v]\rangle\langle l_j[x \mapsto v]\rangle$. From T-Mod, we have $\{\Gamma, x : d\ t; C \vdash^0 e_i : \{\texttt{in}\}\ \texttt{int}\}$, $\Delta(m) = |\,\langle e_i\rangle\,|\,\langle d_j\rangle$, $\{\Gamma, x : d\ t; C \vdash^1 l_j : d'_j t'_j\}$, $\{d_j \subseteq d'_j\}$, and that each t'_j have equal width to the corresponding $t_j\{[x_i \mapsto e_i]\}$ given C. From Lemmas 3 and 4 we get that $\{\Gamma[x \mapsto v]; C \vdash^0 e_i[x \mapsto v] : \{\texttt{in}\}\ \texttt{int}\}$ and $\{\Gamma[x \mapsto v]; C \vdash^1 l_j[x \mapsto v] : d'_j t'_j[x \mapsto v]\}$ therefore using Lemma 2 and T-Mod one more time, we know that $\Delta; \Gamma[x \mapsto v]; C \vdash m\,\langle e_i[x \mapsto v]\rangle\langle l_j[x \mapsto v]\rangle$
- **Case Assign:** P is $\texttt{assign}\ l\ e$ and from the substitution rule $P[x \mapsto v]$ is $\texttt{assign}\ l[x \mapsto v]\ e[x \mapsto v]$. We have two sub-cases to consider:
  - e is an integral value: From T-Assign1, we know that $\Gamma, x : d\ t; C \vdash^1 l : d_1\ t_1$, that t_1 has width i given C. From Lemma 4 we get that $\Gamma[x \mapsto v]; C \vdash^1 l[x \mapsto v] : d_1\ t_1[x \mapsto v]$ Therefore using Lemma 2 with T-Assign1 again, we know that $\Delta; \Gamma[x \mapsto v]; C \vdash \texttt{assign}\ l[x \mapsto v]\ e[x \mapsto v]$.
  - e is anything else: From T-Assign2, we know that $\Gamma, x : d\ t; C \vdash^1 l : d_1\ t_1$, $\Gamma, x : d\ t; C \vdash^1 e : d_2\ t_2$ and that t_1 and t_2 have equal widths given C. From Lemmas 4 and 3 we get that $\Gamma[x \mapsto v]; C \vdash^1 l[x \mapsto v] : d_1\ t_1[x \mapsto v]$, and $\Gamma[x \mapsto v]; C \vdash^1 e[x \mapsto v] : d_2\ t_2[x \mapsto v]$ Therefore using Lemma 2 with T-Assign2 again, we know that $\Delta; \Gamma[x \mapsto v]; C \vdash \texttt{assign}\ l[x \mapsto v]\ e[x \mapsto v]$.
- **Case If:** P is $\texttt{if}\ e\ \texttt{then}\ \langle\langle P_i\rangle\rangle\ \texttt{else}\ \langle\langle P_j\rangle\rangle$ and from the substitution rule $P[x \mapsto v]$ is $\texttt{if}\ e[x \mapsto v]\ \texttt{then}\ \langle\langle P_i[x \mapsto v]\rangle\rangle\ \texttt{else}\ \langle\langle P_j[x \mapsto v]\rangle\rangle$. From T-If, we know that $\Gamma, x : d\ t; C \vdash^0 \{\texttt{in}\}\ \texttt{int}$, that $\{\Delta; \Gamma, x : d\ t; C, e \vdash P_i\}$, and that $\{\Delta; \Gamma, x : d\ t; C, \neg e \vdash P_j\}$. From Lemma 3 we get that $\Gamma[x \mapsto v]; C \vdash^0 e[x \mapsto v] : \{\texttt{in}\}\ \texttt{int}$ and from the induction hypothesis we get that $\{\Delta; \Gamma[x \mapsto v]; C, e \vdash P_i[x \mapsto v]\}$ and $\{\Delta; \Gamma[x \mapsto v]; C, \neg e \vdash P_j[x \mapsto v]\}$ Therefore using T-If again we conclude that $\Gamma[x \mapsto v]; C \vdash \texttt{if}\ e[x \mapsto v]\ \texttt{then}\ \langle\langle P_i[x \mapsto v]\rangle\rangle\ \texttt{else}\ \langle\langle P_j[x \mapsto v]\rangle\rangle$
- **Case For:** P is $\texttt{for}(y = e_1;\ y < e_2;\ y = y{+}e_3)\ \langle P_i\rangle$ and from the substitution rule $P[x \mapsto v]$ is $\texttt{for}(y = e_1[x \mapsto v];\ y < e_2[x \mapsto v];\ y = y + e_3[x \mapsto v])\ \langle P_i[x \mapsto v]\rangle$. From T-For, we know that $\{\Delta; \Gamma, x : d\ t, y : \{\texttt{in}\}\ \texttt{int}; C, y \geq e_1, y < e_2 \vdash P_i\}$, that $\Gamma, x : d\ t, y : \{\texttt{in}\}\ \texttt{int}; C \vdash^0 e_3 : \{\texttt{in}\}\ \texttt{int}$, that $\Gamma, x : d\ t; C \vdash^0 e_1, e_2 : \{\texttt{in}\}\ \texttt{int}$, and that e_3 is positive given C. From Lemma 3 we get that $\Gamma[x \mapsto v], y : \{\texttt{in}\}\ \texttt{int}; C \vdash^0 e_3[x \mapsto v] : \{\texttt{in}\}\ \texttt{int}$, and $\Gamma; C \vdash^0 e_1[x \mapsto v], e_2[x \mapsto v] : \{\texttt{in}\}\ \texttt{int}$ and from the induction hypothesis we get that $\{\Delta; \Gamma[x \mapsto v], y : \{\texttt{in}\}\ \texttt{int}; C, y \geq e_1, y < e_2 \vdash P_i[x \mapsto v]\}$ Therefore using T-For again (with Lemma 2) we know that $\Gamma[x \mapsto y]; C \vdash \texttt{for}(y = e_1[x \mapsto v];\ y < e_2[x \mapsto v];\ y = y + e_3[x \mapsto v])\ \langle P_i[x \mapsto v]\rangle$.

B.2 Proof of Type Preservation

Lemmas 5, 6, 7, 8, and 9 establish that preprocessing preserves typability for e (Lemmas 5 and 6), l, P, and b terms. They are all required to prove Theorem 1

Lemma 5 (Type preservation for level 0 expressions) *If* $\Gamma^+; C \vdash^0 e : d\ t$ *and* $e \overset{0}{\hookrightarrow} e'$ *then* $\Gamma^+; C \vdash^0 e' : d\ t$

Proof We proceed by induction on the evaluation tree of e. We have six cases: cases Id, Index, and Range are vacuously true because none of them can be typed at level 0. T-Par is also vacuously true because Γ^+ by definition does not contain any level 0 variables. In case Int, the lemma hold 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 6 (Type preservation for level 1 expressions) *If* $\Gamma^+; C \vdash^1 e : d\ t$ *and* $e \overset{1}{\hookrightarrow} e'$ *then* $\Gamma^+; C \vdash^1 e' : d\ t$

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

- **Case Id:** e is s and from E-Id e' is s. Therefore the lemma holds trivially.
- **Case Index:** e is $s[e_1]$ and from E-Idx e' is $s[v]$ where $e_1 \overset{0}{\hookrightarrow} v$. From T-Idx we know that $\Gamma^+; C \vdash^0 e_1 : \{\texttt{in}\}\ \texttt{int}$. From Lemma 5, we get that $\Gamma^+; C \vdash^0 v : \{\texttt{in}\}\ \texttt{int}$. Using T-Idx again we get $\Gamma^+; C \vdash^1 s[v] : d\ t$.
- **Case Range:** e is $s[e_1 : e_2]$ and from E-Rg e' is $s[v_1 : v_2]$. Following the same steps of the previous case but using T-Rg instead of T-Idx and using Lemma 5 twice we get the desired result.
- **Case Param:** e is x. The lemma hold vacuously because from T-Par 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 $\{\Gamma^+; C \vdash^1 e_i : d'\ t_i\}$ and using the induction hypothesis on e_i we get $\{\Gamma^+; C \vdash^1 e'_i : d'\ t_i\}$. Using T-Op again, we obtain the required result.

Lemma 7 (Type preservation for LHS values) *If* $\Gamma^+; C \vdash^1 l : d\ t$ *and* $l \overset{1}{\hookrightarrow} l'$ *then* $\Gamma^+; C \vdash^1 l' : d\ t$

Proof This follows directly from Lemma 6 because any l is also an e.

Lemma 8 (Type preservation for parallel statements) *If* $\Delta; \Gamma^+; C \vdash P$ *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* $\{\Delta'; \Gamma^+; C \vdash P'_j\}$

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 $\langle \texttt{module}\ m'\ \langle\rangle\ \langle d_j\ t_j\{[x_i \mapsto v_i]\}\ s_j\rangle\ \texttt{is}\ \langle t_q\{[x_i \mapsto v_i]\}\ s_q\rangle\ \biguplus_k \langle P_r \rangle^{r \in R(k)} \rangle \uplus \biguplus_k \langle D_h \rangle^{h \in H(k)}$. 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 $\Delta(m) = \langle x_i \rangle \langle d_j\ t_j \rangle$, 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 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 7 and 6 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 $\biguplus_k \langle P_r \rangle$, and a sequence of modules $\biguplus_k \langle D_h \rangle$. 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 5 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 1 and 3 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 5 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 9 (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 $\langle\rangle\langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \biguplus_k \langle P_r \rangle^{r \in R(k)}$ 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 8 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 1). The proof works as follows:

Proof Let $p = \langle D_i \rangle m$. From E-Prog, p' is $\langle D_r \rangle \uplus \langle \texttt{module}\ m\ b' \rangle\, m$. 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 9 we can conclude that $\vdash \langle D_r \rangle : \Delta'$ and $\Delta' \vdash b' : M$. This also means that $\langle D_r \rangle \uplus \langle \texttt{module}\ m\ b' \rangle$ is well-typed under the empty environment (T-MSeq). This proves $\vdash p'$.

B.3 Proof of Preprocessing Soundness

Lemmas 10, 11, 12, 13, and 14 establish that the result of preprocessing e (Lemmas 10 and 11), l, P, and b terms respectively produces a fully expanded term of the same category. Theorem 3 establishes this property for circuit descriptions.

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

Lemma 10 (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. Case E-Int0 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 11 (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. Case E-Id1 is immediate since $e' = s$ which is allowed in $\hat{e}$. Cases E-Index1, E-Range1 make use of Lemma 10 to make sure that $e' = s[v]$ and $e' = s[v_1 : v_2]$ are in $\hat{e}$. Case E-Int1 is immediate and Finally case E-Op1 make use of the induction hypothesis to prove its goal.

Lemma 12 (Preprocessing soundness for LHS values) *If* $l \overset{1}{\hookrightarrow} l'$ *then* $l' \in \hat{l}$

Proof Follows directly from Lemma 11

Lemma 13 (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 expanding 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 $\langle \texttt{module}\ m'\ \langle\rangle\ \langle d_j\ t_j\{[x_i \mapsto v_i]\}\ s_j\rangle\ \texttt{is}\ \langle t_q\{[x_i \mapsto v_i]\}\ s_q\rangle \biguplus_k \langle P_r\rangle^{r\in R(k)}\rangle \uplus \biguplus_k \langle D_h\rangle^{h\in H(k)}$ 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 expansion of each $P_k\{[x_i \mapsto v_i]\}$ is a subtree in the expansion 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 by these expansions are in $\hat{D}$ by the inductive hypothesis. Note that we know that every v_i is in $\hat{e}$ from Lemma 10.
- **Case Assign:** From E-Assign, we know that expanding 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 11 and 12 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 expansion result is equal to a sequence of parallel statements and a sequence of modules. These sequences are generated from expanding expanding 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 from both expansions are sound.
  - **Case False:** Similarly it can be seen from E-IfFalse, that expanding each parallel statement in $\langle P_j\rangle$ produces fully expanded terms.
- **Case For:** We have two subcases to consider here:
  - **Case False:** From E-ForFalse, The expansion result is equal to an empty sequence of parallel statements and an empty sequence of modules.
  - **Case True:** From E-ForTrue, The expansion result is equal to a sequence of parallel statements and a sequence of modules. This sequence is generated from expanding the body of the `for` loop after substitution and expanding 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 from both expansions are sound. Again, note that substitutions in E-ForTrue do not introduce any subterms that are not in our expanded 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 10.

Lemma 14 (Preprocessing soundness for main module body) *If* $\langle D_i\rangle \vdash b \overset{1}{\hookrightarrow} b', \langle D_k\rangle$ *then* $b' \in \hat{b}$ *and* $\{D_k \in \hat{D}\}$

Proof From E-Body, we can see that the result from expanding the body b of the main module gives a new body b' equal to $\langle\rangle\langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \biguplus_k \langle P_r\rangle^{r\in R(k)}$ 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 13 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 expansions. Therefore $b' \in \hat{b}$ and $\{D_k \in \hat{D}\}$ are true.

Finally, we prove the Preprocessing Soundness Theorem (Theorem 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 14 we know that for every D_r, we have $D_r \in \hat{D}$ and that $b' \in \hat{b}$, therefore p' is in $\hat{p}$.

C Descriptions of Circuit Families

In this appendix we list the VPP source code of the generic circuits mentioned in this paper. The provided code is purely structural code. It is also standard Verilog code except for optional `where` parameter constraints in the last two examples, and for the `assume` statements. The generated code after preprocessing is Verilog compliant. In fact, it is purely structural Verilog code free from any parameterization or generative constructs.

C.1 Bus Inverter

An N-bit inverter.

```
module invertn (x,y);
  parameter N=4;

  input  [N-1:0] y;
  output [N-1:0] x;
  genvar         i;

  for(i=0;i<N;i=i+1)
    not(x[i],y[i]);
endmodule
```

C.2 Counter

An N-bit synchronous counter. Notice the usage of an assume statement to specify the type of a T flip-flop without having to define it. Assume statements do not exist in standard Verilog and were added for convenience.

```
assume tflipflop(output wire q,input wire t,input wire clk) 9;

module counter(count,next,enable,clk);
  parameter N=4;

  output [N-1:0] count;
  output         next;
  input          enable;
  input          clk;
  wire   [N:0]   t;
  genvar         i;

  assign t[0]=enable;
  for(i=0;i<N;i=i+1) begin
    assign t[i+1] = t[i] & count[i];
    tflipflop tff(count[i],t[i],clk);
```

```
17    end
18  endmodule
```

C.3 Linear Parity Checker

An N-bit parity checker using only 2-bit XOR gates.

```
1   module parity(p,a);
2     parameter N=6;
3
4     input  [0:N-1] a;
5     output         p;
6     wire   [0:N-1] w;
7     genvar         i;
8
9     assign w[0]=a[0];
10    for(i=0;i<N-1;i=i+1)
11      xor (w[i+1],w[i],a[i+1]);
12    assign p = w[N-1];
13  endmodule
```

C.4 Ripple Adder

An N-bit ripple adder.

```
1   module full_adder (sum,cout,a,b,cin);
2     input  a,b,cin;
3     output sum,cout;
4     wire   w1,w2,w3;
5
6     xor(w1,a,b);
7     xor(sum,w1,cin);
8     and(w2,cin,w1);
9     and(w3,a,b);
10    or(cout,w2,w3);
11  endmodule
12
13  module adder(s,cout,a,b,cin);
14    parameter N=8;
15
16    input  [N-1:0] a,b;
17    input          cin;
18    output [N-1:0] s;
19    output         cout;
20    wire   [N:0]   c;
21    genvar         i;
22
23    assign c[0] = cin;
24    for(i=0; i<N; i=i+1)
25       full_adder fa (s[i],c[i+1],a[i],b[i],c[i]);
26    assign cout=c[N];
27  endmodule
```

C.5 Carry Select Adder Block

An N-bit carry select block.

```
1  module full_adder (sum,cout,a,b,cin);
2     input  a,b,cin;
3     output sum,cout;
4     wire   w1,w2,w3;
5
6     xor(w1,a,b);
7     xor(sum,w1,cin);
8     and(w2,cin,w1);
9     and(w3,a,b);
10    or(cout,w2,w3);
11 endmodule
12
13 module ripple_adder(sum,cout,a,b,cin);
14    parameter N=4;
15
16    input  [N-1:0] a,b;
17    input          cin;
18    output [N-1:0] sum;
19    output         cout;
20    wire   [N:0]   cs;
21    genvar         i;
22
23    assign cs[0] = cin;
24
25    for(i=0; i<N; i=i+1)
26      full_adder fa (sum[i],cs[i+1],a[i],b[i],cs[i]);
27
28    assign cout=cs[N];
29 endmodule
30
31 module mux (mout,a,b,sel);
32   output mout;
33   input  a,b,sel;
34
35   wire w1,w2;
36   or (mout,w1,w2);
37   and (w1,a,~sel);
38   and (w2,b,sel);
39 endmodule
40
41 module carry_select_adder_block(sum,cout,a,b,cin);
42    parameter N=4;
43
44    input  [N-1:0] a,b;
45    input          cin;
46    output [N-1:0] sum;
47    output         cout;
48    wire   [N-1:0] sum0,sum1;
49    wire           cout0,cout1;
50    genvar         i;
51
52    ripple_adder #(N) r1 (sum0,cout0,a,b,0);
53    ripple_adder #(N) r2 (sum1,cout1,a,b,1);
54
55    for(i=0; i<N; i=i+1)
56      mux sum_mx (sum[i],sum0[i],sum1[i],cin);
```

```
57
58    mux cout_mx (cout,cout0,cout1,cin);
59 endmodule
```

C.6 Decoder

An N-by-2^N decoder. Notice the use of a single parameter instead of two parameters as was required before the introduction of Verilog's power operator $**$. If two parameters are used, a user of this module might instantiate it with incompatible parameter values essentially generating an incorrect circuit. This is also the first example to illustrate multi-dimensional arrays.

```
1  module decoder (dec_out,dec_in);
2     parameter N=2;
3
4     input  [N-1:0]     dec_in;
5     output [2**N-1:0] dec_out;
6     wire   [N-1:0]     ndec_in;
7     wire   [N-1:0]     temp [2**N-1:0];
8     genvar             i,j;
9
10    for(i=0;i<N;i=i+1)
11      not (ndec_in[i],dec_in[i]);
12
13    for(i=0;i<2**N;i=i+1) begin
14       for(j=0;j<N;j=j+1) begin
15    if ((i>>j) % 2==0)
16      assign temp[i][j]=ndec_in[j];
17    else
18      assign temp[i][j]=dec_in[j];
19      end
20       assign dec_out[i] = &temp[i];
21    end
22 endmodule
```

C.7 Multiplexer

A 2^N-by-1 multiplexer. Again, notice the usage of assume statements to declare a decoder without having to redefine it and the usage of one parameter for the multiplexer's input and its selection lines. This example illustrates a parameter constraint using a `where` construct that does not exist in Verilog.

```
1  assume decoder #(N 2)
2   (output wire [2**N] a,
3    input wire [N] b) 2**N+N;
4
5  module gen_mux(mux_out,mux_in,sel);
6    parameter M=3 where M >= 1;
7
8    input [2**M-1:0] mux_in;
9    input [M-1:0]    sel;
10   output           mux_out;
11   genvar           i;
12   wire  [2**M-1:0] decsel;
```

```
13    wire  [2**M-1:0] p;
14
15    decoder #(M) dec1 (decsel,sel);
16    for(i=0;i<2**M;i=i+1)
17      and (p[i],decsel[i],mux_in[i]);
18    assign mux_out = |p;
19  endmodule
```

C.8 Multiplier

A purely combinational circuit to multiply an N-bit unsigned integer by an M-bit unsigned integer. This circuit features two uncorrelated parameters.

```
1   assume adder #(N 4)
2    (output wire [N] sum,
3     output wire count,
4     input wire [N] a,
5     input wire [N] b,
6     input wire cin) 5*N;
7
8   module multiplier(c,a,b);
9      parameter N=6 where N >= 2;
10     parameter M=3 where M >= 1;
11
12     input  [N-1:0]   a;
13     input  [M-1:0]   b;
14     output [N+M-1:0] c;
15     wire   [N-1:0]   tmpand [M-1:0];
16     wire   [N-1:0]   tmpsum [M-1:0];
17     wire       tmpc [M-1:0];
18     genvar       i,j;
19
20     for(i=0;i<N;i=i+1)
21       for(j=0;j<M;j=j+1)
22         and (tmpand[j][i],a[i],b[j]);
23
24     assign c[0] = tmpand[0][0];
25
26     assign tmpsum[0] = tmpand[0];
27     assign tmpc[0] = 0;
28
29     for(i=0;i<M-1;i=i+1) begin
30        adder #(N) add1 (tmpsum[i+1],
31             tmpc[i+1],
32             tmpand[i+1],
33             {tmpc[i],tmpsum[i][N-1:1]},
34             0);
35        assign c[i+1] = tmpsum[i+1][0];
36     end
37
38     assign c[N+M-1:M] = {tmpc[M-1],tmpsum[M-1][N-1:1]};
39  endmodule
```