RICE UNIVERSITY

Static Analysis for Circuit Families

by

Cherif R. Salama

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

Prof. Walid Taha, Chair
Assistant Professor of Computer Science,
Rice University

Prof. Robert Cartwright
Professor of Computer Science, Rice
University

Prof. Joseph Cavallaro
Professor of Electrical and Computer
Engineering, Rice University

Dr. John O'Leary
Principal Engineer, Intel Corporation

Houston, Texas
May, 2010
ABSTRACT

Static Analysis for Circuit Families

by

Cherif R. Salama

As predicted by Gordon Moore, the number of transistors on a chip has roughly doubled every two years. Microprocessors featuring over a billion transistors are no longer science fiction. For example, Intel's Itanium 9000 series and Intel's Xeon 7400 series of processors feature 1.7 and 1.9 billion transistors respectively. To keep up with the emerging needs of contemporary very large scale integration (VLSI) design, industrial hardware description languages (HDLs) like Verilog and VHDL must be significantly enhanced. This thesis pinpoints some of the main shortcomings of the latest Verilog standard (IEEE 1364-2005) and shows how to overcome them by extending the language in a backward compatible way.

To be able to cope with more complex circuits, well-understood higher-level abstraction mechanisms are needed. Verilog is already equipped with promising generative constructs making it possible to concisely describe a family of circuits as a parameterized module; however these constructs suffer from two problems: First,
their expressivity is limited and second, they are not adequately supported by current tools. For instance, there are no static guarantees about the properties of the description generated as a result of instantiating a generic description with particular parameter values.

Addressing both problems while remaining backward compatible led us to select a statically typed two-level languages (STTL) formal framework. By formalizing a core subset of Verilog as an STTL, we were able to define a static type system capable of: 1) checking the realizability of a description, 2) detecting bus width mismatches and array bounds violations, and 3) providing parametric guarantees on the resources required to realize a generic description. The power of the chosen framework is once more demonstrated as it also allows us to enrich the language with a new set of constructs that are designed to be expanded away when instantiated.

To experiment with these ideas we implemented VPP, a Verilog Preprocessor with a built-in type checker. VPP is an unobtrusive tool accepting extended Verilog descriptions but generating descriptions compatible with any tool compliant with the Verilog standard.

Our experience throughout this research showed that STTLs present a particularly suitable framework to formalize and implement generative features of a language.
Acknowledgments

Although a Ph.D. dissertation is officially submitted as being the work of an individual, I believe this is never actually the case. A candidate usually owes much of his achievements to many people whose support was invaluable. This is particularly true in my case and I want to use this space to sincerely thank all the persons who have contributed to this work or have helped me through my journey at Rice.

My deepest gratitude goes first to my supervisor Prof. Walid Taha whose continuous guidance, support, and encouragement are the reason this thesis exists today. His direct and indirect contributions to this work are countless. Working with him has been an honor and also an enjoyable experience.

I am deeply indebted to Dr. Jim Grundy and Dr. John O’Leary with whom I have had the privilege to collaborate on several papers. They have both been very supportive, patient, and always provided professional advice and guidance.

My special thanks to Prof. Corky Cartwright, my first programming languages teacher and a priceless resource on everything related to programming. Prof. Cartwright’s knowledge, expertise, and enthusiasm are contagious.

I would also like to express my gratitude to all my thesis committee members and to Prof. Joseph Cavallaro in particular for taking the time to evaluate this work and for providing valuable comments and suggestions.
I was also extremely lucky to have co-authored papers with brilliant colleagues including Gregory Malecha, Angela Zhu, and Jeniffer Gillenwater. I would like to thank them all for their hard work and for the enjoyable times we spent together. I would also like to thank Yilong Yao for being the first user of VPP who has not participated in its implementation.

On two consecutive summers, I had the opportunity to go for an internship at IBM's research labs in Austin. Both summers, I was reporting to Dr. Andrew Martin who has patiently tutored me and showed me many programming tricks that I now use frequently.

I had many fruitful discussions about hardware description languages and VLSI design with Yousra Alkabani and Ahmed Elnably. On several occasions, these conversations have helped me re-organize my thoughts and my research priorities.

I am also thankful to all the members of the Resource Aware Programming group at Rice, especially Dr. Edwin Westbrook, Dr. Ron Garcia, Mathias Ricken, Jun Inoue, Dr. Raj Bandyopadhyay, Fulong Chen, Travis Martin, and Marisa Peralta. I consider myself very fortunate to have interacted with such talented and kind people.

I would like to thank all my friends for being there for me whenever I needed them. In particular, I would like to thank Cherif Boctor, Dr. Sami Kirolos, his wife Heba Nazih, Ahmed Khattab, Tamer Ali, Mohammad Youssof, Nader Wadie, Dr. Wael Abd El-Galiel, John Kiriazi, Aly Farahat, Seif El-Dawlatly, Sameh Zakhary,

Words cannot express how grateful I am to my parents and my sister who have supported and encouraged me every single day since I was born. I would have never had the courage or the ambition to go this far without them. I am also very grateful to my in-laws for their understanding and continuous support.

Finally, I would like to thank my lovely fiancée, Nermine for being who she is and for her unconditional love to me. I am so grateful to have her in my life. I dedicate this thesis to her.

**Funding:** My Ph.D. program was generously supported by Rice University, by the National Science Foundation (NSF) SoD award 0439017, and by Intel Corporation through the Semiconductor Research Corporation (SRC) Task ID 1403.001. I am very grateful to all four organizations.
Contents

Abstract iii
Acknowledgments v
List of Figures xv
List of Tables xvii

1 Introduction 1

1.1 The Design Productivity Gap 2

1.1.1 Leveraging Block Reuse: Circuit Families 3

1.2 The VLSI Design Flow 4

1.2.1 Specifications and Design 4

1.2.2 Synthesis and Optimization 8

1.2.3 Placement and Routing 11

1.2.4 Fabrication/Programming 11

1.3 This Thesis and The Design Flow 11

1.3.1 Assessment of Current Description Levels 12

1.3.2 Generative Constructs 14

1.4 Limitations of Verilog's Generative Constructs 14

1.4.1 Delayed Typing of Generative Constructs 15

1.4.2 Hidden Width Inconsistencies 15
1.4.3 Difficulty of Design Space Exploration ................. 16
1.4.4 Expressivity Limitation ................................. 16
1.5 Contributions .............................................. 17
  1.5.1 Formalization and Static Checking of Generative Constructs . 17
  1.5.2 Detecting Bus Width and Array Bounds Violations .......... 18
  1.5.3 Parametric Gate Count Analysis ........................ 18
  1.5.4 Increasing Verilog's Generative Power .................. 19
  1.5.5 The Verilog Preprocessor (VPP) ........................ 19

2 Formalization and Static Checking of Generative Constructs .... 21
  2.1 Verilog 2005 Generative Constructs ...................... 21
    2.1.1 Limitations ......................................... 23
  2.2 Contributions ........................................... 26
  2.3 Syntax of Featherweight Verilog (FV) and Its Synthesizable Subset . 28
    2.3.1 Notational Conventions ................................ 29
    2.3.2 Formal Syntax in Backus-Naur Form (BNF) ............. 29
    2.3.3 Simplifications to Verilog Syntax ................... 31
    2.3.4 FV Example .......................................... 32
    2.3.5 Synthesizable Subsets .............................. 33
  2.4 Formal Specification of Operational Semantics for Preprocessing .. 34
  2.5 Formal Specification of Type System .................... 40
2.5.1 Typing Environments ........................................ 40
2.5.2 Typing Rules .................................................... 41
2.5.3 Additional Synthesizability Conditions ...................... 44
2.6 Technical Results ................................................. 44
  2.6.1 Type Preservation ............................................. 44
  2.6.2 Type Safety ..................................................... 45
  2.6.3 Preprocessing Soundness .................................... 45
2.7 Abstractions in Practice .......................................... 46
2.8 Implementation .................................................. 46
  2.8.1 Performance Evaluation ...................................... 48
2.9 Summary .......................................................... 49

3 Detecting Bus Width and Array Bounds Violations ............. 52
  3.1 Verilog 2005 Padding Semantics .............................. 52
  3.2 Contributions .................................................... 60
  3.3 Approach ........................................................ 61
    3.3.1 Dependent Types ........................................... 61
    3.3.2 Recording Control Flow Information ..................... 62
    3.3.3 Generating Consistency Constraints ..................... 62
    3.3.4 Verifying Consistency .................................... 63
    3.3.5 Unreachable Code Detection ............................... 65
3.4 SMT Solver Usage ........................................ 66
3.5 Extensions to Featherweight Verilog ................. 67
  3.5.1 FV Syntax in BNF .................................. 67
  3.5.2 Formal Specifications of FV Type System .......... 69
  3.5.3 Formal Specifications of FV Operational Semantics .. 71
3.6 Technical Results ...................................... 73
  3.6.1 Type Preservation .................................. 73
  3.6.2 Type Safety ....................................... 74
  3.6.3 Preprocessing Soundness ............................ 75
3.7 Related Work .......................................... 75
3.8 Implementation ........................................ 78
  3.8.1 Limitations ....................................... 79
  3.8.2 Performance Evaluation ............................ 79
3.9 Summary ................................................ 81

4 Parametric Gate Count Analysis .......................... 83
  4.1 Design Space Exploration ............................. 83
  4.2 Contributions ........................................ 84
  4.3 Approach ............................................. 85
  4.4 Formal Account ....................................... 87
    4.4.1 Additions to Type System ......................... 87
4.4.2 Summation Closed Forms .......................... 90
4.4.3 Correctness ........................................... 91
4.5 Analysis for Other Resources ......................... 94
4.6 Related Work ............................................ 94
4.7 Implementation .......................................... 95
4.7.1 Experimental Evaluation ............................ 96
4.8 Summary .................................................. 98

5 Increasing Verilog’s Generative Power ............... 99
  5.1 Contributions ........................................... 99
  5.2 Missing Language Features ............................ 100
  5.3 Proposed Syntax ....................................... 101
  5.4 Semantics of Proposed Extensions ................... 104
  5.5 Static Analyses ....................................... 106
    5.5.1 Termination Analysis .............................. 108
  5.6 Implementation ....................................... 113
    5.6.1 Experimental Results ............................ 113
  5.7 Summary .............................................. 114

6 Conclusion ................................................. 115
  6.1 Recap of Results ..................................... 115
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1</td>
<td>For Chapter 2</td>
<td>140</td>
</tr>
<tr>
<td>D.1.1</td>
<td>Substitution Lemma</td>
<td>140</td>
</tr>
<tr>
<td>D.1.2</td>
<td>Type Preservation</td>
<td>141</td>
</tr>
<tr>
<td>D.1.3</td>
<td>Type Safety</td>
<td>147</td>
</tr>
<tr>
<td>D.1.4</td>
<td>Preprocessing Soundness</td>
<td>147</td>
</tr>
<tr>
<td>D.2</td>
<td>For Chapter 3</td>
<td>151</td>
</tr>
<tr>
<td>D.2.1</td>
<td>Substitution Lemma</td>
<td>151</td>
</tr>
<tr>
<td>D.2.2</td>
<td>Type Preservation</td>
<td>156</td>
</tr>
<tr>
<td>D.2.3</td>
<td>Type Safety</td>
<td>162</td>
</tr>
<tr>
<td>D.2.4</td>
<td>Preprocessing Soundness</td>
<td>162</td>
</tr>
</tbody>
</table>
List of Figures

1.1 The Standard VLSI Design Flow (greatly simplified) ............. 5
1.2 Hardware Description Abstraction Levels ....................... 6

2.1 A 4-bit Ripple Adder ........................................ 22
2.2 Substitution .................................................. 37
2.3 Operational Semantics ....................................... 50
2.4 Type System .................................................. 51

3.1 A 4-bit Synchronous Counter using T Flip-Flops ............... 54
3.2 Type-checker Interaction with an SMT Solver ................... 67
3.3 Type System For Parallel Statements .......................... 72
3.4 Type System For All Other Constructs ........................ 73
3.5 Substitution .................................................. 74
3.6 Operational Semantics ....................................... 82

4.1 Type System for Parallel Statements .......................... 91
4.2 Type System For All Other Constructs ......................... 92
4.3 Summation Identities ....................................... 93

5.1 The Ripple Pattern Connecting \( n \) Instances of \( \text{circ} \) (adapted from [1]) 101
5.2 Instantiation Graph for the Ripple Pattern ...................... 110
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3 Propagating Parameter Values Through The Loop</td>
<td>111</td>
</tr>
<tr>
<td>5.4 A Cycle in the Instantiation Graph Indicating Mutually Recursive</td>
<td>112</td>
</tr>
<tr>
<td>C.1 Operational Semantics Errors For Parallel Statements</td>
<td>138</td>
</tr>
<tr>
<td>C.2 Operational Semantics Errors For All Other Constructs</td>
<td>139</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Impact of Abstraction on Code Size (in lines) ..................... 47
2.2 VPP's Type Checking Performance ...................................... 48
3.1 VPP's Performance ....................................................... 80
4.1 Parametric Gate Count For Circuit Families ......................... 97
Chapter 1
Introduction

As predicted 45 years ago by Gordon Moore [2], the number of transistors on a chip has roughly doubled every two years [3]. Microprocessors featuring over a billion transistors are no longer science fiction. For example, Intel's Itanium 9000 series and Intel's Xeon 7400 series of processors feature 1.7 and 1.9 billion transistors respectively [4]. Despite the plethora of computer aided design (CAD) tools available, building digital circuits of comparable sizes or even orders of magnitudes smaller, known as very large scale integration (VLSI) circuits, remains an engineering challenge. A rapidly increasing design productivity gap exists between the progress rate of manufacturing technology and the progress rate of industrial hardware description languages (HDLs) like Verilog [5] and VHDL [6] that have barely evolved since their first introduction in the mid 80s. My thesis is that

*To bridge the design productivity gap, industrial HDLs must be significantly enhanced.*

To achieve this goal, this thesis proposes a set of static analysis techniques for a Verilog subset. These analyses collectively aim to provide practical support for correctly writing parameterized circuit descriptions that we refer to as circuit families. Additionally, this thesis proposes a set of backward compatible Verilog extensions
intended to push the envelope of parameterization higher allowing more complex
circuit families to be described. Although we focus our attention to Verilog, most of
the ideas presented in this thesis are equally applicable to VHDL.

1.1 The Design Productivity Gap

The international technology roadmap for semiconductors (ITRS) [7] defines the
design productivity gap as,

\textit{The number of available transistors growing faster than the ability to
meaningfully design them. This gap impacts IC [integrated circuit] product
value, placing at risk foundry amortization, return-on-investment (ROI)
for supplier industries, and indeed the entire semiconductor investment
cycle. ... Failure to effectively develop and deploy the roadmap of DT [de-
sign technology] innovations will break the long-standing trend of progress
in the semiconductor industry. Hence we view these DT gaps as crises
that must be addressed in the next 15 years.}

Increasing design productivity is crucial for keeping up with the industry require-
ments. The ITRS identifies design block reuse as one of the design requirements and
states that,

\textit{Pre-designed IP [intellectual property] blocks promise to allow assembling
SoCs [systems-on-chip] of unprecedented complexity in a very short time.}
Design block reuse is the process of using pre-designed hardware modules as building blocks when designing a new circuit. We believe that if leveraged, block reuse can play a significant role in bridging the productivity gap.

1.1.1 Leveraging Block Reuse: Circuit Families

Building circuits using a library of pre-designed fixed modules is the most basic form of code reuse in Verilog. To maximize reusability, library modules should be customizable to fit in a wide variety of circuits. For example, a 32-bit adder is not very useful when designing a circuit that needs a 16-bit adder. On the other hand, a customizable-width adder is useful not only in this situation but whenever an adder is needed. Such a customizable adder is called a parameterized adder and represents a family of circuits (in this case a family of adders) as opposed to a single circuit. In general a parameterized module represents a circuit family whose individual members correspond to instantiations with specific parameter values. We also use the terms generic description and circuit template to refer to parameterized modules.

Verilog is already equipped with promising constructs making it possible to concisely describe a family of circuits as a parameterized module. These constructs are known as generative constructs because a circuit family can be viewed as a template for generating various concrete instances. However, generative constructs tend to be avoided by hardware designers due to their limitations. This thesis provides the necessary techniques and tools to overcome these limitations. We discuss these limi-
tations and the specific contributions of our work in more depth towards the end of this chapter (and throughout the whole thesis), but first to help the reader understand where exactly in the process our work fits, we briefly explain the standard steps involved in VLSI design also known as the VLSI design flow.

1.2 The VLSI Design Flow

We present a greatly simplified design flow with just enough details to familiarize the reader with the process. For a more detailed understanding of the design flow we direct the reader to textbooks on the subject [8,9].

Building a digital circuit starts with an idea of a circuit having a certain functionality and ends with an integrated circuit (IC) realizing it. The final product is either a custom made silicon chip known as an application-specific integrated circuit (ASIC) or an implementation on a general purpose field-programmable gate array (FPGA). As opposed to ASICs, FPGAs are reconfigurable ICs and offer a cheaper alternative for prototyping or when mass production is not intended. In both cases the design process goes through a similar set of phases interleaved with simulation and/or verification. Figure 1.1 summarizes the process.

1.2.1 Specifications and Design

The first step in the process is to express the circuit concept to implement as a set of (potentially informal) specifications and then to convert these specifications
into a hardware description. There are three levels of details that can be used to do so. The higher the level of abstraction, the fewer details are provided, making the description simpler and shorter at the expense of having less control over the resulting circuit structure and its properties. A designer is not required to provide all three descriptions. The choice of whether to provide one, two, or all three levels depends on several factors including what he wants to do with the description (verification, simulation, and/or manufacturing), how much time is available to him, and how much control over the resulting circuit he wants to have. Other factors include the tools available to him, the circuit libraries that he has access to, and his personal expertise and preferences.

The following sections explains the different levels of abstractions and Figure 1.2
summarizes them. Since our contributions mainly affect how descriptions are written, we come back to this topic after presenting the rest of the design flow.

**Algorithmic Level Description**

The highest level of abstraction is to specify the functionality of a circuit at an algorithmic level. At this level, the designer is not concerned by any of the actual implementation details. Such a description serves both as a specification of the circuit and as a model that can be simulated and verified. The description itself in this case looks a lot like code written in an imperative general purpose programming language like C. In fact C [10], C++ [11], and SystemC [12] are among the most popular algorithmic HDLs. Verilog, its extension SystemVerilog [13], and VHDL allow this style of programming through their behavioral subsets.
Register Transfer Level Description

The next level of abstraction is more concerned about the structure of the circuit although it does not pin down all the details. This level specifies how the various control and data signals travel between different parts of the circuit. In particular the number of state-holding elements (memories, flip-flops) is fixed but the connections between them are described using various constructs, including conditionals as well as boolean and arithmetic equations. The notion of timing is also made explicit. A description providing this level of detail is usually known as a register transfer level (RTL) description. Both Verilog and VHDL can express RTL descriptions using a combination of structural and restricted behavioral constructs.

Gate Level Description

The lowest abstraction level is a purely structural description in which the building blocks of a module and their interconnections are fully specified. The building blocks can either be primitives (gates and flip-flops) or other previously defined modules.

While it is possible to create a purely structural description by drawing a circuit schematic, this approach is only practical for small circuits. For large circuits, gate level descriptions are usually written using the purely structural subset of Verilog or that of VHDL.
1.2.2 Synthesis and Optimization

Hardware synthesis refers to the set of processes required to go from a description of a circuit to a realizable specification known as a netlist. A netlist is a list of interconnected library cells. The library cells depend on the target technology and as such the synthesis process will also vary depending on whether we are targeting an ASIC or an FPGA implementation. A description is said to be synthesizable if it can be automatically translated to a netlist using a synthesis tool. Unfortunately, this definition implies that synthesizability depends on the particular synthesis tool used.

Since there are several levels of descriptions providing various levels of details, the synthesis process heavily depends on the level of the input description as explained below.

High Level Synthesis

It is possible to automate the transition from an algorithmic description to an RTL. This process is called High Level Synthesis (HLS). Since a behavioral description leaves out many details, this process involves many choices. For example, an HLS process needs to decide how many storage units and computation units of each type to allocate. It also needs to bind the operations appearing in the description to the allocated resources and schedule them. For more details about HLS synthesis we refer the reader to Sharp's dissertation reprint [14].
HLS is still a young research topic; however, there are already many commercial tools available that are capable of performing it including Catapult C [15], C-to-Silicon Compiler [16], PICO C Synthesis [17], AutoESL [18], C-to-Verilog [19], and Symphony [20] (which accepts high level descriptions written in Matlab [21]).

**Logic Synthesis**

Logic synthesis is the process of converting an RTL description to a gate level description. This step involves converting several RTL expressions and constructs into explicit primitive gates and storage elements. RTL descriptions are usually regarded as synthesizable descriptions although this is not entirely true. The IEEE Standard for Verilog RTL synthesis [22] gives a list of rules and guidelines explaining how to write RTL code supported by compliant synthesis tools. An RTL description committing even a subtle violation of these guidelines might not be synthesizable. A similar IEEE standard exists for VHDL as well [23].

Logic synthesis is a mature field of research and is the basis of many popular commercial tools including Design Compiler [24], Encounter RTL Compiler [25], Leonardo Spectrum [26], ISE [27], and the free Icarus Verilog Compiler [28].

**Technology Mapping**

Technology mapping, also known as library binding, is the process of converting a gate level description to a netlist using only the library cells of the targeted technology.
This step is simple to achieve and is usually done as part of logic synthesis in case the description was originally provided at a higher level. Technology mapping is optionally (but frequently) preceded and followed by various optimizations.

**Optimization**

Synthesis algorithms often attempt to minimize the area, delay, and (more recently) power consumption of a description. For example, for combinational circuits, minimization can optionally be used to remove redundant gates from a gate level design or a netlist, generating an optimal or near optimal circuit providing the same functionality. If optimal circuits are required, Karnaugh maps [29] or the Quine-McCluskey algorithm [30, 31] can be used. The latter is not restricted to functions of at most 6 input variables and is easily amenable to computer implementation, but is still exponential in the number of input variables. In practice, heuristics are employed; for example, Berkeley’s Espresso algorithm [32] and tools exploiting it can obtain near-optimal circuits without any particular restrictions on the number of input variables.

The end product of the synthesis process is a netlist optimized for a specific target technology that can be passed to the next step of the design process.
1.2.3 Placement and Routing

In case the end product is an ASIC, the step following synthesis is the creation of the physical layout of the silicon wafer. This step involves several tasks including partitioning the system into various building blocks, floor planning, placement, and routing collectively known as placement and routing.

In case of an FPGA, a similar notion of placement and routing is also required to map the netlist obtained from synthesis to the logic blocks of the FPGA. The result is a file known as a bitstream.

1.2.4 Fabrication/Programming

The last step for an ASIC is to send the files generated by placement and routing to the fabrication facility to be manufactured based on the target technology. These files will be used to create the photomasks needed to create the physical ASIC.

For the FPGA, the bitstream file is directly used to program the FPGA using a specially designed programmer or a computer. The FPGA is then automatically configured to achieve the required functionality.

1.3 This Thesis and The Design Flow

As explained in the previous section, there are several non-trivial steps required to build a circuit. The good news is that most of these steps are mechanizable using various Computer Aided Design (CAD) tools. Of course this does not apply
to the design step where a description of the circuit needs to be provided. Once a satisfactory description exists, it can be fed to a series of tools that will eventually end up generating the necessary files to realize the physical circuit. In this thesis we focus on the circuit description process, its limitations, and how to address them in a practical way.

1.3.1 Assessment of Current Description Levels

Providing a satisfactory description in a productive way remains a challenge. By satisfactory, we mean a description that is both functionally correct and synthesizable into a circuit respecting the design constraints (such as timing, manufacturing cost, and power consumption). Looking at the different levels of abstractions that are available today we see that none of them is ideal for this task for the following reasons:

• **Algorithmic Level Descriptions**: Using this level of description, designers can be very productive (assuming that they are familiar with C-like languages which is not necessary true for hardware designers), but these descriptions can be very hard or just impossible to synthesize. Even when this is not the case, HLS may create very inefficient circuits. Moreover, the original description is so distant from the resulting circuit that it is very hard to analyze any of the circuit properties until after it is synthesized. Finally, the hardware designer usually has very little control over the structure of the circuits being generated.
by the HLS process which can render him helpless if the resulting circuit fails to meet the design constraints.

- **RTL Descriptions:** RTL descriptions are currently the most popular way of writing synthesizable hardware descriptions. They offer a reasonable level of abstraction (although much lower than algorithmic descriptions) and can be synthesized into reasonably efficient hardware for the most part. Nevertheless, the designer does not have full control over the structure of the final circuit. Another disadvantage is that RTL descriptions are not suitable for describing circuit families, which as pointed out earlier, is important in order to increase the hardware designer's productivity.

- **Gate Level Descriptions:** Descriptions at this level are synthesizable since they only need to be mapped to the target technology cells. Additionally, gate level descriptions offer the designer full control. However, even if hierarchical descriptions allow for some code reuse, these descriptions are too low level. Coding at this level can be compared to programming in assembly language in the software world. Also as in the case of RTL, this level of description is not particularly suited to describing circuit families.
1.3.2 Generative Constructs

Both Verilog and VHDL offer code generation constructs allowing concise description of circuit families. These constructs are built on top of purely structural constructs providing a higher level of abstraction than gate level descriptions while retaining the same level of control over the structure of the circuit. Generative constructs are designed to be expanded away during a phase known as elaboration. If successful, the result of elaboration is a purely structural (gate level) description.

Unfortunately, these constructs suffer from two severe problems making their usage impractical in most settings: First, their expressivity is limited and second, they are not adequately supported by current tools. For instance, there are no static guarantees about the properties of the description generated as a result of instantiating a generic description with particular parameter values. In the next sections we explore these limitations in more depth and explain how we overcome them. We discuss the lack of static guarantees first because we had to address this issue before we could think of increasing the generative constructs' expressivity (making the already hard problem of static checking harder).

1.4 Limitations of Verilog's Generative Constructs

In this section, we present the main shortcomings of Verilog's generative constructs that we address in this thesis.
1.4.1 Delayed Typing of Generative Constructs

In Verilog the constructs `generate/endgenerate` are used to describe regular or parameterized hardware designs. When used effectively, they can make hardware descriptions shorter, more understandable, and more reusable. Although these constructs are fully-expanded during elaboration, it can be difficult to understand and predict the properties of the circuit descriptions they generate. Is the generated code even type safe? Is it synthesizable? What physical resources (e.g. combinatorial gates and flip-flops) does it require? Does answering these questions depend on the parameter values in case of circuit families? It is often impossible to answer these questions without first generating the fully-expanded code and even then we cannot answer the last question. Finally, when used incorrectly, generative constructs can even fail to elaborate for some or all possible parameter values.

1.4.2 Hidden Width Inconsistencies

Verilog has padding semantics that allow designers to write descriptions where wires of different bit widths can be interconnected; however, many of these 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 \texttt{generate} blocks to describe circuit families only makes the situation worse as it hides such inconsistencies making them harder to detect. Inconsistencies in the generated code are only exposed after elaboration, when the code is fully-expanded.

1.4.3 Difficulty of Design Space Exploration

Given the large design space and the strict time constraints imposed by a tight time-to-market, circuit designers often have a hard time coming up with the best possible design. In many cases, they just go with the first working idea that they find. If given the means to quickly explore the consequences of changing a module's design on the whole circuit's properties, they will certainly have a better chance of creating better products. For digital circuits, one such property is the number of gates consumed by a certain design. This number is directly correlated with the circuit's footprint, its manufacturing cost, and its power consumption. Currently these properties can only be computed for a given circuit instance (as opposed to a family of circuits) and only after a time-consuming synthesis process.

1.4.4 Expressivity Limitation

Despite the availability of generative constructs, they are limited to parameterized modules, conditionals, and loops. Using these constructs, it is often impossible to express circuits with more complicated structures or highly parameterized design.
This limitation significantly impedes the usefulness of such constructs and discourages hardware designers from attempting to use them simply because they do not provide a level of abstraction that is sufficiently higher than gate level descriptions.

1.5 Contributions

In this thesis we dedicate our efforts to make generative constructs more useful so that they can provide a predictable (controllable) abstraction mechanism.

1.5.1 Formalization and Static Checking of Generative Constructs

Our first (and perhaps most important) contribution is a formal framework allowing us to deal with generative constructs and their elaboration in a disciplined way (Chapter 2). By viewing Verilog as a statically typed two-level language (STTL), we are able to reflect the distinction between values that are known at elaboration time and values that are part of the circuit computation. This distinction is crucial for determining whether generative constructs, such as iteration and module parameters, are used in a synthesizable manner. This framework allows us to develop a static type system that guarantees synthesizability. The type system achieves safety by performing additional checks on generative constructs. To illustrate this approach, we develop a core calculus for Verilog that we call Featherweight Verilog (FV) and an associated static type system. We formally define a preprocessing step analogous to the elaboration phase of Verilog, and characterize the kinds of errors that can
occur during this phase. Finally, we show that a well-typed design cannot cause preprocessing errors, and that the result of its elaboration is always a synthesizable circuit.

1.5.2 Detecting Bus Width and Array Bounds Violations

We show that inconsistencies such as mismatched bus widths and array bounds violations can be pinned down prior to elaboration using static analysis (Chapter 3). We combine dependent types and constraint generation to reduce the problem of detecting inconsistencies to a satisfiability problem. Once reduced, the problem can be handed over to 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 extend FV's type system and prove that a well-typed FV description will always elaborate into an inconsistency-free description.

1.5.3 Parametric Gate Count Analysis

We present a method to statically compute the post-elaboration gate count of a parameterized module as a function of its parameter values (Chapter 4). Again, the method relies on extending FV's type system further to include gate count expressions in the inferred module signatures. The method is formalized and proven correct.
1.5.4 Increasing Verilog's Generative Power

All the previous contributions are concerned with dealing with the deficiencies of Verilog's existing generative constructs or are trying to enhance their analysis in order to increase their usability. While the existing constructs can be quite useful for describing some relatively simple circuit patterns, they are not expressive enough to describe more complex designs. We deal with these expressivity limitations by identifying the key language extensions that can be used to overcome them (Chapter 5). We also show how the static analyses we proposed can be generalized to accommodate them.

1.5.5 The Verilog Preprocessor (VPP)

Current CAD tools provide limited support (if any) for existing generative constructs and naturally they do not support our newly introduced constructs. Luckily, all the generative constructs are meant to be removed during the elaboration phase which can be seen as a preprocessing step or, otherwise stated, as a source to source transformation on Verilog code. The result of this transformation is standard, purely structural, Verilog code that is free from any generative constructs.

In order to experiment with the presented ideas and to make them of practical use, we implemented a Verilog Preprocessor (VPP). VPP has a built-in type checker capable of performing all the analyses described earlier to prove that a generic de-
scription is well-typed and to statically estimate the number of gates it requires as a function of its parameters.

VPP is an unobtrusive tool because it can easily fit in the VLSI design flow. Instead of describing the desired circuit (or circuit family) at an algorithmic, RTL, or gate level; the designer should use generative constructs. The next step in the process would be to type check it using VPP. By default if the description is found to be well-typed, VPP will also elaborate it into a gate level description written in standard Verilog which can then be used in the rest of the VLSI design flow.

The VPP implementation of each of the proposed analyses and extensions is presented near the end of the corresponding chapter. Appendix B provides instructions on installing and using VPP.

This research has led to several important insights and technical results that were published in [33–36].
Chapter 2
Formalization and Static Checking of Generative Constructs

This chapter introduces Verilog's generative constructs and the main difficulties that arise when using them. This is followed by a formalization that provides a framework for dealing with these difficulties. This chapter lays the foundations needed to be able to deal with generative construct in a disciplined way.

2.1 Verilog 2005 Generative Constructs

The Verilog language has three kinds of constructs. Constructs of the first kind used to provide a gate level description. These are usually referred to as structural constructs. Constructs of the second kind allow for an algorithmic level description. These are most often used for simulation purposes, and are usually referred to as behavioral constructs. When used carefully behavioral and structural constructs can be combined to provide RTL descriptions. The third kind of construct describes the generation of more Verilog code through the elaboration process. Constructs of this kind are known as generative constructs, and they include parameterized modules, conditionals and iteration. Generative constructs provide an abstraction mechanism on top of the gate level by allowing for compact and reusable descriptions of circuit families.
For example, a family of adders such as the one presented in Figure 2.1 can be described as follows:

```verilog
module adder(s,cout,a,b,cin);
  parameter N=4;
  input [N-1:0] a,b;
  input cin;
  output [N-1:0] s;
  output cout;
  wire [N:0] c; genvar i;

  assign c[0] = cin;
  generate
    for(i=0; i<N; i=i+1)
      full_adder fs (s[i],c[i+1],a[i],b[i],c[i]);
  endgenerate
  assign cout = c[N];
endmodule
```

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

![Figure 2.1 A 4-bit Ripple Adder](image)

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

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

### 2.1.1 Limitations

The most important property of a description is whether or not it is physically realizable as a circuit. This property is often referred to as \text{synthesizability}. Intuitively, a synthesizable description is a description that has a direct correspondence to a circuit. Unfortunately, the line between synthesizable and non-synthesizable descriptions is unclear and \text{ad hoc}. The Verilog Register Transfer Language (RTL) synthesis standard [22] does not formally draw this line. Instead, it gives some synthesizability
guidelines, illustrated by a series of good and bad examples. Although this is useful to understand what kind of descriptions should be synthesizable, it is not sufficient.

The first Verilog standard issued in 1995 [37] supported iterations and conditionals only as behavioral statements. Whether these constructs were synthesizable or not was implementation dependent. The Verilog-2001 standard [38], and subsequent Verilog-2005 [39] and SystemVerilog [13] standards, extend Verilog-95 with the `generate/endgenerate` construct, which allows conditionals and single variable iteration statements to appear in parallel statements. When enclosed in `generate` blocks, these statements are elaborated into ordinary structural statements prior to simulation or synthesis.

Because current Verilog compilers analyze `generate` blocks only after they are elaborated, errors remain undetected until synthesis, when they are more difficult to diagnose. In addition, only concrete instances of a family of designs — like the default 4-bit wide instance of the ripple adder above — are checked for errors. A similar problem is familiar in programming languages that introduce a stage of code generation before execution. Macros in C and templates in C++ are examples where the characteristics of a program cannot be understood without "expanding away" the macros or templates. For this reason sophisticated use of such features is wisely curtailed by developers despite the obvious power of the technique. So it is with Verilog: In both educational and industrial settings it is common to see the loop in
the previous generic design manually unrolled into the following concrete instance:

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

Alternatively, designers commonly use scripting languages, such as Perl [40], to generate Verilog code for specific instances of module families. Both manual unrolling and scripting are undesirable. Unrolling is tedious, error-prone, and non-scalable. Scripting employs a language that manipulates hardware descriptions as strings rendering static analysis of any kind impractical. The generated code is not even guaranteed to be syntactically correct!

A third popular alternative is using a circuit generator software. These generators are programs with a built-in library of circuit families that a designer can use. This approach does not suffer from the problems described above; however, it suffers from the more fundamental issue that these programs are closed. They do not allow the designer to create new circuit families. Additionally, they do not give the designer any control over how the circuits he requested are designed.

However, more disciplined approaches exist: Two-level [41, 42] and multi-level languages [43, 44] have been studied as a way to understand software code generation. They provide a formal infrastructure that allows characteristics of programs to be checked without requiring expansion. For example, Kiselyov, Swadi and Taha [45] have shown how to generate highly optimized, type correct Fast Fourier Transform
kernel routines from compact algorithmic descriptions written using a multi-level language. Similarly, Taha, Ellner and Xi [46] have shown how to generate heap bounded implementations of sorting programs from a compact, parameterized sorting algorithm written using a two-level language. In both cases all static analysis is performed prior to code generation.

Bluespec SystemVerilog (BSV) [47] uses Term Rewriting Systems (TRS) [48] to provide powerful generate-like features with static checks. However, the relationship between BSV and TRS is not formally explained. A formal semantics is a prerequisite for having static guarantees.

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

2.2 Contributions

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

- Provides a proof of concept that we can check properties of the circuit generated from elaboration without actually performing elaboration and paves the way for more aggressive static checking (demonstrated in the following chapters).
• Suggests that designers may be able to use high level abstractions without sacrificing the benefits of static checking. For example, any misuse of abstractions that might impair synthesizability will be detected statically.

• Enables the use of a single, tightly integrated language to describe circuits and circuit families.

• Enables the use of the same type checker to check descriptions before and after elaboration.

• Enables checking the synthesizability of families of circuits once, rather than repeating the check for each instantiation.

To achieve static checking, we provide rigorous definitions for two notions of synthesizability, namely, *obvious synthesizability* and *general synthesizability* in an implementation independent way (Section 2.3.5). These concepts allow us to pin down the Verilog constructs that are interesting from the synthesizability point of view and formally treat them.

To address the above goals from a semantic point of view, we define Featherweight Verilog (FV), a calculus for a representative core of structural Verilog (Section 2.3). A two-level operational semantics for FV captures how various generative constructs should be elaborated and what can go wrong during this process (Section 2.4). A two-level type system is used to define the conditions needed to guarantee that various
constructs do not interfere with synthesis (Section 2.5).

We establish three properties of FV (Section 2.6). Theorem 1 states that elaboration preserves typing. Theorem 2 states that it is always safe to perform elaboration of a well-typed design by showing that elaboration never depends on wire values. Theorem 3 states that the result of elaboration is obviously synthesizable. Combined, these results establish that a well-typed design is synthesizable, which implies that we can statically check for synthesizability of a description before elaborating it using a relatively simple type checker. Auxiliary results and a complete formal proof for each of these theorems appear in Appendix D.

A prototype implementation in the form of a Verilog preprocessor (VPP) was developed. VPP statically checks the synthesizability of a Verilog description (possibly containing high level abstractions) and elaborates it if it is well-typed (Section 2.8).

2.3 Syntax of Featherweight Verilog (FV) and Its Synthesizable Subset

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

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

- A sequence is either the empty sequence () or a non-empty sequence \( h :: t \) with a head \( h \) and a tail sequence \( t \).

- We write \( (X_i)^{i \in I} \) 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 \( (X_i) \) instead of \( (X_i)^{i \in I} \).

- We write \( X \oplus Y \) for the concatenation of the two sequences \( X \) and \( Y \).

- We write \( \bigcup_{k} (D_r)^{r \in R(k)} \) for the concatenation of all \( (D_r)^{r \in R(k)} \).

- We write \( |S| \) for the length of the sequence \( S \).

2.3.2 Formal Syntax in Backus-Naur Form (BNF)

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

\[
\begin{align*}
\text{Module} & \quad m \in \text{ModuleNames} \\
\text{Signal} & \quad s \in \text{IdentifierNames} \\
\text{Elaboration Variable} & \quad x, y \in \text{ParameterNames} \\
\text{Operator} & \quad f \in O \\
\text{Index} & \quad h, i, j, k, q, r \in N \\
\text{Index Domain} & \quad H, I, J, K, Q, R \subseteq N
\end{align*}
\]

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

\[
\begin{align*}
\text{Circuit Description} & \quad p ::= (D_{i})^{i \in I} \ B \\
\text{Module Definition} & \quad D ::= \text{module } m \ b \\
\text{Module Body} & \quad b ::= (x_{i})^{i \in I} (d_{j} s_{j})^{j \in J} \text{ is } (t_{k} s_{k})^{k \in K} (P_{r})^{r \in R} \\
\text{Direction} & \quad d \subseteq \{\text{in, out}\} \\
\text{Type} & \quad t \in T ::= \text{wire | int} \\
\text{Parallel Statement} & \quad P ::= m (e_{i})^{i \in I} (l_{j})^{j \in J} | \text{assign } e \\
& \quad | \text{if } c \text{ then } (P_{1})^{i \in I} | \text{else } (P_{1})^{i \in I} \\
& \quad | \text{for } (y = c ; c ; y = c) (P_{1})^{i \in I} \\
\text{LHS value} & \quad l ::= s | s[e] | s[e : c] \\
\text{Expression} & \quad e ::= l \mid x \mid v \mid f(e_{i})^{i \in I} \\
\text{Value} & \quad v ::= (0 \mid 1)^{32}
\end{align*}
\]

A circuit description $p$ is a sequence of module definitions $(D_{i})^{i \in I}$ followed by a module name $m$. The module name indicates which module from the preceding sequence of definitions represents the overall input and output of the system. A module definition consists of a name $m$ and a module body $b$. A module body itself consists of four sequences: (1) module parameter names $(x_{i})^{i \in I}$, (2) port declarations (carrying direction, and name for each port) $(d_{j} s_{j})^{j \in J}$, (3) local variable declarations (carrying type, and name for each variable) $(t_{k} s_{k})^{k \in K}$, and (4) parallel statements $(P_{r})^{r \in R}$. A port direction indicates whether the port is input, output, or bidirectional. The type of a local variable can be either $\text{wire}$, if the variable represents an internal physical connection, or $\text{int}$ otherwise. A parallel statement can be a module instantiation, an $\text{assign}$ statement, a conditional statement, or a $\text{for}$-loop. A module instantiation specifies module parameters $(e_{i})^{i \in I}$ as well as port connections $(l_{j})^{j \in J}$. An $\text{assign}$ statement consists of a left hand side (LHS) value $l$ and an expression $e$. An LHS
value is either a variable \( s \), an array lookup \( s[e] \), or an array range \( s[e : e] \). An expression is either an LHS value \( l \), a parameter name \( x \), a 32-bit integer \( v \), or an operator application \( f(e_i)^{iej} \).

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

\[
X \quad ::= \quad p \mid D \mid b \mid P \mid l \mid e
\]

### 2.3.3 Simplifications to Verilog Syntax

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

- All sequences are represented uniformly as \( <a, b, \ldots, z> \).

- The start of the module body is marked with the terminal "is".

- Local variable declarations are aggregated immediately after the is terminal.

- Module parameters are listed as arguments rather than being declared locally.

- Module parameters do not have default values, so values must always be provided explicitly with each use.

- The direction of a port is declared as part of the formal module argument (as allowed starting from Verilog-2001), rather than in the body of the module definition (as in Verilog-95).
• Variable direction is represented using a set instead of a keyword. The sets
\{in\}, \{out\}, or \{in, out\} replace the keywords input, output, and inout
respectively. The last set is also used for non-directional variables such as
internal signals.

• The if-statement always has an else clause, but that clause can contain an
empty sequence of statements.

• Wire sizes are dropped from terms because Verilog uses automatic coercions to
pad arrays of wires of different sizes to match them.

• All for-loop variables are declared implicitly and are local to the loop.

• Primitive gates are not explicitly modeled, but they can be expressed using
logical operators.

• Integers are represented in binary.

2.3.4 FV Example

The ripple adder module presented in the introduction can be written using the

```verilog
calculus as follows:

module adder <N> <out s, out cout, in a, in b, in cin> is
<wire c>
<assign c[0] cin,
for(i=0; i<N; i=i+1)
    < full_adder <> <a[i],c[i+1],a[i],b[i],c[i]> >,
assign cout c[N]>
```
As explained earlier, there are no default values for parameters and no wire sizes in the calculus. This syntax is more concise and more suitable for formal treatment.

2.3.5 Synthesizable Subsets

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

- **obvious synthesizability** means that a description uniquely determines a directed graph where nodes are either primitive gates or obviously synthesizable modules and edges are wires connecting them.

- **general synthesizability** (or synthesizability for short) means that a description is either obviously synthesizable or will become obviously synthesizable after elaboration.

Note that uniqueness of the graph does not imply a unique hardware implementation (because there are different libraries and even different ways to implement a circuit using a given library). Instead, it means that there is a systematic and deterministic way to convert the description to a graph representing the circuit. These definitions are applicable to any hardware description language in general and are implementation independent. Thus descriptions that are obviously synthesizable according to this definition should be synthesizable by all sensible synthesis tools supporting the language in which the description is written.
For Verilog, when structural descriptions are free of high level abstractions, they are obviously synthesizable. The same applies to FV as well since it is a subset of structural Verilog: Well-formed FV descriptions free from abstraction (parameterized modules, conditionals, and for-loops) are obviously synthesizable. An FV description is well-formed if it is syntactically correct and satisfies a few conditions that are captured by our type system as defined in section 2.5.

### 2.4 Formal Specification of Operational Semantics for Preprocessing

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

---

*Despite the fact that the term preprocessing is often used to refer to rather ad hoc tools like the C preprocessor (cpp), we prefer to use it for our highly disciplined approach because it implicitly conveys the light weight, unobtrusive quality that VPP is designed to achieve.
We use a big-step operational semantics [49] indexed by the level of the computation to formally specify the preprocessing phase. The specification dictates how preprocessing should be performed, what the form of the preprocessed circuit descriptions should be, and what errors can occur during preprocessing.

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

\[ \text{Possible Term} \quad X_{\perp} ::= X \mid \text{err} \]

This allows us to write \( p_{\perp} \) or \( e_{\perp} \) to denote a construct that may either be the constant \( \text{err} \) or a value from \( p \) or \( e \), respectively.

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

\[
\begin{align*}
\text{Preprocessed Term } \hat{X} = & \{ u \mid u \in X_{\perp} \land \forall Y. (Y \in \text{subterms}(u) \Rightarrow \\
(Y = (x_{i})_{i \in I} \mid (d_{j} s_{j})_{j \in J} \mid a \mid t_{k} y_{k})_{k \in K} \mid (P_{r})_{r \in R} \Rightarrow I = \emptyset) \\
\land (Y = m \mid (e_{i})_{i \in I} \mid (l_{j})_{j \in J} \Rightarrow I = \emptyset) \\
\land (Y = s[e_{1} : e_{2}] \Rightarrow e_{1} \in v \land e_{2} \in v) \\
\land (Y = s[e] \Rightarrow e \in v) \\
\land (Y \neq \text{if } e \text{ then } (P_{1})_{i \in I} \text{ else } (P_{2})_{i \in J}) \\
\land (Y \neq \text{for}(y = e; c; y = e)(P_{1})_{i \in I}) \} \}
\end{align*}
\]

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

- A module definition has no parameters.
• Module instantiations do not pass any parameters.

• Array indices and array bounds are 32-bit values.

• There are no if-statements.

• There are no for-loops.

A notion of substitution is also required. \( X[x \mapsto v] \) denotes the substitution of \( x \) by \( v \) in \( X \). A principal challenge in designing preprocessing systems is the avoidance of accidental variable capture, which occurs when the binding occurrence of a variable is changed. Systems such as the C preprocessor (cpp) are seen as fragile because they do not avoid accidental capture. Preprocessing systems that avoid accidental variable capture are called hygienic [50]. The key to hygienic preprocessing is to employ a notion of substitution that respects the binding structure of variables – using, for example, free and bound variable conventions as in the lambda calculus [51].

In FV, preprocessing only substitutes level 0 variables (parameters and for-loop indices) with their corresponding integral values. Since values are distinct from variables, we do not need to worry about accidental variable capture. Therefore, defining the substitution rules as shown in Figure 2.2 is a straightforward process. Additionally, since the Barendregt convention is not hiding any complexity, the implementation follows naturally. The only thing to be aware of is the distinction between level 1 variables that should not be affected by the substitution and level 0 variables that
might be. This distinction can easily be made by recording the level of each variable in the abstract syntax tree while traversing the description.

\[ P[x \mapsto v] \]
\[
m(e_i)(l_i)[x \mapsto v] = m(c_2[x \mapsto v])(l_i[x \mapsto v])
\]
\[
(\text{assign } l e)[x \mapsto v] = \text{assign } l[x \mapsto v] e[x \mapsto v]
\]
\[
(\text{if } c \text{ then } \langle P_1 \rangle \text{ else } \langle P_2 \rangle)[x \mapsto v] = \text{if } c[x \mapsto v] \text{ then } \langle P_1[x \mapsto v] \rangle \text{ else } \langle P_2[x \mapsto v] \rangle
\]
\[
(\text{for}(y = e_1; e_2; y = e_3)(P_i))[x \mapsto v] = \text{for}(y = e_1[x \mapsto v]; e_2[x \mapsto v]; y = e_3[x \mapsto v])(P_i[x \mapsto v])
\]

\[ l[x \mapsto v] \]

\[ e[x \mapsto v] \]
\[
s[x \mapsto v] = s
\]
\[
s[e[x \mapsto v]] = s[e[x \mapsto v]]
\]
\[
s[e_1; e_2][x \mapsto v] = s[e_1[x \mapsto v]; e_2[x \mapsto v]]
\]
\[
x[x \mapsto v] = v
\]
\[
y[x \mapsto v] = y \text{ if } y \neq x
\]
\[
v'[x \mapsto v] = v'
\]
\[
f(e_i)[x \mapsto v] = f(e_i[x \mapsto v])
\]

\textbf{Figure 2.2} Substitution

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

Successful preprocessing is defined in Figure 2.3. Preprocessing a circuit description (E-Prog) starts by preprocessing the main module and other modules are instantiated as needed. Preprocessing the body of the main module (E-Body) involves preprocessing each of its statements. The curly braces surrounding the preprocessing judgement indicate that this is a set of judgements of size $|K|$, one for each parallel statement $P_k$. Preprocessing a statement $P_k$ can generate several statements $(P'_k)_{r \in R(k)}$ and can involve instantiating several modules $(D_h)_{h \in H(k)}$. The set $R(k)$ is the set of indices of the resulting statements. This set is a function of $k$ since it will be different for each of the parallel statements preprocessed. Similarly $H(k)$ is the set of indices of the resulting modules. All the obtained statements and modules are aggregated (in order) and returned.

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

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

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

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

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

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

This example describes a module `badinv` with one output port `q` and two input ports `a` and `n`. This circuit is supposed to inverts each of the input bits of `a`, connecting the result to the corresponding output bit in `q`. The `~` operator is Verilog's inversion operator. Clearly this circuit cannot be realized because the number of inverters composing it is determined by the value on the input wire `n`. In the next section, we define the type system that allows us to statically detect such violations.
2.5 Formal Specification of Type System

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

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

2.5.1 Typing Environments

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

- **Module Type**
  \[ M ::= k \langle d_i \rangle_{i \leq i} \]
- **Operator Signatures**
  \[ \Sigma \in \Pi. \mathbb{O} \times \mathbb{N} \times \mathbb{T} \rightarrow \mathbb{T} \]
- **Level**
  \[ n ::= 0 | 1 \]
- **Module Environment**
  \[ \Delta ::= \Gamma \mid (m : M) :: \Delta \]
- **Variable Environment**
  \[ \Gamma ::= \Gamma \mid (s : d t) :: \Gamma \mid (x : d t) :: \Gamma \]
- **Level 1 Variable Env.**
  \[ \Gamma^+ ::= \Gamma \mid (s : d t) :: \Gamma^+ \]

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

2.5.2 Typing Rules

Figure 2.4 defines the rules for the judgment ⊢ p which specifies when a circuit
description p is well-typed. The typing rules formalize the following requirements.
A circuit description p is typable when the declarations it contains produce a valid
module environment and the main module has a type that involves no module param-
eters (T-Prog). This means that all the modules are well-typed and the top module
is not parameterized since this module is automatically instantiated (recall that in
FV, parameters do not have default values).

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

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

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

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

By specifying which expressions need to be typable at level 0, these typing rules guarantee the static availability of all the information needed to get rid of the abstractions during elaboration. By doing so, the type system guarantees the success of elaboration for all well-typed descriptions as well as the success of its synthesis as will be shown in Section 2.6.

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

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

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

Both issues are orthogonal to the problems addressed by our type system and we expect that they can easily be checked by other techniques. For instance, in the next chapter we explain how to add restrictions on for-loops to ensure termination. A linear type system can be used to avoid multiple assignment but we chose to avoid the extra associated complexity in this presentation.

2.6 Technical Results

We establish three theorems whose complete proofs are presented in Appendix D.

2.6.1 Type Preservation

We first show that the preprocessing of a well-typed description produces a well-typed description. Formally:

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

*Sketch*. The proof proceeds by induction on the derivation of the second judgment.
2.6.2 Type Safety

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

**Theorem 2** (Type Safety). If \( \vdash p \) and \( p \overset{1}{\rightarrow} p' \) then \( p' \neq \text{err} \)

**Proof.** This result follows directly from Theorem 1 since no typing rules will consider \text{err} well-typed. \(\square\)

2.6.3 Preprocessing Soundness

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

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

**Sketch.** The proof proceeds by induction on the derivation of the first judgment. \(\square\)

As stated in Section 2.3.5, well-typed FV programs free from abstractions are obviously synthesizable. Combining this observation with the last three theorems, we see that we can use our type system to check for the synthesizability of a circuit description statically prior to elaboration: If an FV program is well-typed, Theorem 2 says its elaboration will not produce an error and Theorems 1 and 3 say that the result
will be well-typed and abstraction-free. The result of elaborating a well-typed FV program is therefore obviously synthesizable.

2.7 Abstractions in Practice

We manually refactored several industrial hardware descriptions from [52, 53] to use generative constructs. The manual re-factoring was straightforward and mainly involved replacing blocks of instances with loops or nested loops. The hardest part was to figure out the increment expressions for each of the loop indices. Comparing the re-factored code to the original shows that using abstraction can cut the number of lines in half, as depicted in Table 2.1. Of course, being multipliers, these circuits are highly regular and therefore are particularly suitable to show the usefulness of the abstractions we are talking about. But designing multipliers is still a formidable engineering challenge where engineers use all the help they can get to make the task more tractable.

The results included here are preliminary experimental results that demonstrate that using abstractions is valuable in practical examples not only for the simple circuits such as ripple adders presented earlier.

2.8 Implementation

VPP is an open source prototype implementation of the ideas we presented throughout this thesis. VPP is entirely written in OCaml [54]. It uses Ocamlex and Ocamly-
Table 2.1  Impact of Abstraction on Code Size (in lines).

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Original</th>
<th>Using</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenRISC 1200's 32x32 multiplier [52]</td>
<td>2538</td>
<td>1405</td>
<td>44.6%</td>
</tr>
<tr>
<td>OpenSPARC T1's 64x64 multiplier [53]</td>
<td>2510</td>
<td>1167</td>
<td>53.5%</td>
</tr>
</tbody>
</table>

acc for parsing Verilog programs. Appendix B explains how to download and install VPP. This section summarizes the main results from our experience with implementing and using the ideas proposed in this chapter.

VPP includes a type checker based on the typing rules defined in this chapter. If the description contains generative constructs, VPP’s type checker determines whether they are used in a synthesizable manner. If the given description is proved synthesizable, then it is elaborated into an equivalent, obviously synthesizable description using the defined preprocessing rules.

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

2.8.1 Performance Evaluation

VPP’s type checker is based on a type system. As such it is a modular checker and therefore is expected to be reasonably efficient. Table 2.2 shows the time required to type check the Verilog examples mentioned above along with the code size for each example. It also shows the time required to elaborate the re-factored versions. These experiments were conducted on a machine with the following specifications: MacBook running Mac OS X version 10.5.6, 2 GHz Intel Core 2 Duo, 4 MB L2 cache, 2 GB 667 MHz DDR2 SDRAM.

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

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Code Size (lines)</th>
<th>TC (msec)</th>
<th>TC rate (lines/msec)</th>
<th>Elaboration (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original 32x32 mult.</td>
<td>2538</td>
<td>19.4</td>
<td>130.8</td>
<td>-</td>
</tr>
<tr>
<td>Original 64x64 mult.</td>
<td>2510</td>
<td>25.8</td>
<td>97.2</td>
<td>-</td>
</tr>
<tr>
<td>Re-factored 32x32 mult.</td>
<td>1405</td>
<td>8.3</td>
<td>169.2</td>
<td>643</td>
</tr>
<tr>
<td>Re-factored 64x64 mult.</td>
<td>1167</td>
<td>8.7</td>
<td>134.1</td>
<td>1237</td>
</tr>
</tbody>
</table>
2.9 Summary

This chapter lays the foundation for dealing with Verilog’s generative constructs in a formal way. It provides the first static guarantee about a circuit family description which is its synthesizability. In the next chapter we build on top of this to provide more guarantees. In particular we show how the type system can be extended to detect wire width inconsistencies and array bounds violations that might be hidden by Verilog’s padding semantics and by generative constructs.
Figure 2.3  Operational Semantics
\[ \Gamma \vdash p \quad \vdash (D_i) : \Delta \quad \Delta(m) = 0 \quad (d_i) \quad (T-\text{Prog}) \]

\[ \Delta \vdash (D_i) : \Delta \quad \Delta \vdash \{ j : [] \} : (T-\text{MEmpty}) \quad \Delta \vdash b : M \quad (m : M) :: \Delta \vdash (D_i) : \Delta' \quad (T-\text{MSeq}) \]

\[ \{ d_j \neq \emptyset \} \quad \{ \Delta; \Gamma \vdash P_1 \} \quad \Delta = \langle x_i : \{ \text{in} \} \ \text{int} \rangle \ \cup \ \langle s_j : \text{d}_{j} \ \text{wire} \rangle \ \cup \ \langle s_k : \{ \text{in} \ \text{out} \} \ \text{t}_k \rangle \]

\[ \Delta \vdash \langle x_i \rangle / \langle d_j \rangle / \langle s_k \rangle / \langle P'_1 \rangle : \langle (x_i) \rangle / \langle d_j \rangle / \langle s_k \rangle / \langle P'_1 \rangle (T-\text{Body}) \]

\[ \Gamma \vdash \langle l_j : d_j, t_j \rangle \quad \{ d_j \subseteq d'_j \} \quad \Delta; \Gamma \vdash m(e_i) (l_j) \quad (T-\text{Mod}) \]

\[ \Gamma \vdash e : \{ \text{in} \} \ \text{int} \quad \Delta; \Gamma \vdash l : d_1 t_1 \quad \text{in} \in d_2 \quad \Gamma \vdash e : d_2 t_2 \quad (T-\text{Assign}) \]

\[ \Delta; \Gamma \vdash \text{assign } l \ e \]

\[ \Gamma \vdash \langle \text{if } e \ \text{then} (P_1) \ \text{else} (P_2) \rangle \quad (T-\text{If}) \]

\[ \Gamma, y : \{ \text{in} \} \ \text{int} \vdash 0 \quad e_1, e_2 : \{ \text{in} \} \ \text{int} \quad \Gamma \vdash e_1 : \{ \text{in} \} \ \text{int} \quad \Gamma \vdash e_2 : \{ \text{in} \} \ \text{int} \quad \Delta; \Gamma \vdash \text{for} (y = e_1; e_2 y = e_3)(P_i) \quad (T-\text{For}) \]

\[ \Gamma \vdash \langle l^n t : d t \rangle \quad \text{See } \Gamma \vdash \langle l^n t : d t \rangle \]

\[ \Gamma \vdash \langle s = d \ t \rangle \quad \Gamma \vdash \langle s : d t \rangle \quad \Gamma \vdash \langle l^n c : \{ \text{in} \} \ \text{int} \rangle \quad (T-\text{Id}) \]

\[ \Gamma \vdash \langle s = d t \rangle \quad \Gamma \vdash \langle s[e] : d t \rangle \quad (T-\text{Idx}) \]

\[ \Gamma \vdash \langle e = d t \rangle \quad \Gamma \vdash \langle e_1, e_2 : \{ \text{in} \} \ \text{int} \rangle \quad \Gamma \vdash \langle s[e_1] e_2 : d t \rangle \quad (T-\text{Rg}) \quad \Gamma \vdash \langle x : \{ \text{in} \} \ \text{int} \rangle \quad (T-\text{Par}) \]

\[ \Gamma \vdash \langle u : \{ \text{in} \} \ \text{int} \rangle \quad (T-\text{Int}) \quad \{ \Gamma \vdash \langle e_i : d t_i \rangle \} \quad (T-\text{Op}) \quad \Gamma \vdash \langle f(e_i) : \{ \text{in} \} \ \text{int} \rangle \quad \Sigma(\langle e_i \rangle)(f, n, \langle t_i \rangle) \]

Figure 2.4  Type System
Chapter 3
Detecting Bus Width and Array Bounds Violations

This chapter identifies another potential source of design errors which is Verilog’s padding semantics. It also extends the formalism in the previous chapter to guarantee that such errors are statically detected.

3.1 Verilog 2005 Padding Semantics

Hardware designers are reluctant to adopt generative constructs because current tools do not provide enough support to use them safely. Despite the fact that generative constructs only make sense in the context of synthesizable circuits, current tools do not attempt to statically check for synthesizability, let alone inconsistencies that Verilog tolerates due to its padding semantics. Most tools will accept a description, elaborate it, and then synthesize it into a graph of connected components known as a netlist. If anything goes wrong in the 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 required.

To illustrate Verilog’s padding semantics and the kind of errors they can hide, let’s 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, if the circuit is naively synthesized, an extra inverter will be used to invert the most significant input bit and then its output will be discarded.

Let's consider now a more interesting example: A 4-bit synchronous counter. The
counter receives two inputs: a clock \( \text{clk} \) and an enable signal \( \text{en} \). If \( \text{en} \) is high (equal to 1), then the 4-bit output \( \text{count} \) is incremented at each clock. The next output can be used to create a wider counter by connecting it to the \( \text{en} \) port of another counter thus cascading both. The schematic diagram in figure 3.1 shows how to construct such a counter out of \( T \) flip-flops and and gates.

![Schematic Diagram]

**Figure 3.1** A 4-bit Synchronous Counter using \( T \) Flip-Flops

This module can be described using structural Verilog as follows:

```verilog
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);
  tflipflop tff_2 (count[2], t[2], clk);
  tflipflop tff_3 (count[3], t[3], clk);
  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 3.1. The result is the following parameterized module using a generative loop:

```verilog
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;
    generate
        for(i = 0; i < N; i = i + 1) begin
            tfliphop tff (count [i], t [i], clk);
            assign t [i + 1] = t [i] & count [i];
        end
    endgenerate
    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 the generic version an off-by-one error causing array bounds violations can be
easily introduced if the hardware designer inadvertently wrote the for-loop condition as \( i \leq N \). 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 they would only do so 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).

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 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 untiill elaboration to detect inconsistencies is not a good strategy for the following reasons:

- Elaboration can potentially be time consuming. Static analysis could save us 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. Therefore the designer can only get 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. Statically checking a generic module before elaboration will provide guarantees about all its possible instantiations. Once this is done, this module can be safely used in any design without any further checks. On the other side, elaboration errors might not show up depending on the parameter values used when instantiating a module. 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 of the last point, 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. What does this tell us about
this module? Does it violate array bounds or not? Without enough static checking, the answer is: "It depends"! This is clearly undesirable.

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 cause the description to be rejected. In case 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} in case he really wants 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 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 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 represents a family of circuits, the tool’s target should be to verify that this is the case for all possible instantiations (for all possible parameter values).
3.2 Contributions

We provide 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 3.3). We also show how the same framework allows us to detect and reject "meaningless" unreachable code (Section 3.3.5). This ability follows naturally from the constraint gathering approach we have to do in order to detect other inconsistencies.

We formalize our approach by embedding it in a powerful type system. To do so we extend the syntax of Featherweight Verilog (FV) to allow for explicit array width declarations (Section 3.5.1). We extend FV's type system to use dependent types and to enforce the required constraints (Section 3.5.2).

We restate the properties of our formalization: 1) Type preservation, 2) Type safety, and 3) Preprocessing soundness (Section 3.6) and prove that they still hold. But since our type systems rejects inconsistencies, these three theorems combined not only guarantee that a well-typed FV description will always elaborate into a description with no generative constructs remaining, but also guarantee that the resulting description will be inconsistency-free. The complete proofs of these theorems
are in Appendix D.

Finally we discuss the implementation of these ideas in VPP (Section 3.8).

3.3 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.

3.3.1 Dependent Types

A dependent type is a type that depends on a value [55,56]. 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.
3.3.2 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. This restriction also makes it possible for the type system to guarantee the termination of elaboration.

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.

3.3.3 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's 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.

### 3.3.4 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 $\langle c_i \rangle \triangleright c$.

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 be handed over to a standard SMT solver.

For example, let's take the $i \leq \text{max}(N, 0)$ 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: $\forall i, N.(i < N \land i \geq 0) \Rightarrow i \leq \text{max}(N, 0)$. We can prove this by proving the unsatisfiability of its negation: $\exists i, N.(i < N \land i \geq 0 \land i > \text{max}(N, 0))$. 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.
3.3.5 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:

```verbatim
module adder (sum, a, b);
    parameter N = 8;
    input [N-1 : 0] a;
    input [N-1 : 0] b;
    output [N : 0] sum;
    generate
        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);
        endgenerate
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.

3.4 SMT Solver Usage

To check for satisfiability of constraints, we rely on an SMT solver. An SMT solver 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, an SMT solver is invoked to verify they are still satisfiable. In case they are 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, we pass its negation along with the set of collected constraints to the SMT solver 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 3.2 illustrates the interaction between a type checker implementing these ideas and an SMT solver. Given an unelaborated description, the type checker generates a bunch of integer satisfiability problems that are handed over to an SMT solver. The type checker finally decides the well-typedness of the description based on its own typing rules and the set of responses from the solver.
3.5 Extensions to 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 this chapter we extend FV definitions to allow us to statically check for inconsistencies. In the following subsections we present the required extensions to 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.

3.5.1 FV Syntax in BNF

In this section we reprise the FV grammar definition from the previous chapter with minor modifications. The meta-variables are as defined in Section 2.3.
\[ p ::= (D_t)^{t \in T} m \]
\[ D ::= \text{module } m b \]
\[ b ::= (x_i)^{i \in I} (d_j, t_j, s_j)^{j \in J} \text{ is } (l_k, s_k)^{k \in K} (P_t)^{t \in T} \]
\[ d \subseteq \{\text{in}, \text{out}\} \]
\[ t \in T ::= \text{wire} | \text{int} | f(e, e) \]
\[ P ::= m (e_i)^{i \in I} (l_j)^{j \in J} | \text{assign } l \ e \]
\[ | \text{if } e \text{ then } (P_i)^{i \in I} \text{ else } (P_j)^{j \in J} \]
\[ | \text{for } (y = c; y < e; y = y + e) (P_i)^{i \in I} \]
\[ l ::= s | s[e] | s[e : e] \]
\[ e ::= l | x | v | f(e_i)^{i \in I} \]
\[ v ::= \langle 0 | 1 \rangle^+ \]

The main difference from the syntax defined in Section 2.3 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 types explicit makes FV closer to the original Verilog syntax. We had omitted those previously because they were not pertinent to synthesizability. 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. The first expression does not necessarily correspond to the lower bound. It might correspond to the upper bound as well. 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 and to help prove their termination. 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.

The general term $X$ is still used to range over all syntactical constructs. We also use the same standard notation that we used in the formal treatment of FV in the previous chapter.

### 3.5.2 Formal Specifications of FV Type System

To define the type system, once again we need the following auxiliary notions:

- **Module Type** $M ::= \langle x_i \rangle_{i \in I} \langle d_j \rangle_{j \in J}$
- **Operator Signatures** $\Sigma \in \Pi \cdot \mathcal{O} \times \mathcal{N} \times \mathcal{T}^\varnothing \rightarrow \mathcal{T}$
- **Level** $n ::= 0 | 1$
- **Module Environment** $\Delta ::= \{ m : M :: \Delta \}$
- **Variable Environment** $\Gamma ::= \{ s : d t :: \Gamma \}$
- **Level 1 Variable Env.** $\Gamma^+ ::= \{ s : d t :: \Gamma^+ \}$
- **Constraint Environment** $C ::= \{ c :: C \}$
- **Constraint** $c ::= e R e | -c$
- **Relational Operator** $R ::= < | > | \leq | \geq | = | \neq$

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.

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 $\text{width}$ that returns the width given type information:

\[
\begin{align*}
\text{width}(\text{int}) &= \infty \\
\text{width}(\text{wire}) &= 1 \\
\text{width}(t(e_1, e_2)) &= (e_2 - e_1 + 1) \cdot \text{width}(t)
\end{align*}
\]
Note that width of int is considered to be infinite (or unbounded). This means that integer variables like 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 \textit{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 \textit{order}:

\[
\begin{align*}
\text{order}(\text{int}) &= \text{int} \\
\text{order}(\text{wire}) &= \text{wire} \\
\text{order}(t(e_1, e_2)) &= \text{order}(t)(\text{min}(e_1, e_2), \text{max}(e_1, e_2))
\end{align*}
\]

For types that do not have bounds, \textit{order} is just the identity function. Otherwise \textit{order} makes sure that the lower bound (which the minimum of the given bounds) is presented before the upper bound (which is the maximum of the given bounds).

Figures 3.3 and 3.4 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 the previous chapter 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.

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 parallel statements is shown in Figure 3.3. 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:

\[
\text{Satisfiable Constraint Environment} \quad C ::= \quad [\_ | c :: C]
\]

A satisfiable constraint environment cannot contain conflicting constraints.

### 3.5.3 Formal Specifications of FV Operational Semantics

In the previous chapter 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.
\[
\begin{align*}
&\{\Gamma; C \vdash_0 e_1 : \text{int}\} \quad \Delta(m) = \left\langle e_2, d_j t_j \right\rangle \\
&\{\Gamma; C \vdash_1 i_j : d_j' t_j'\} \quad \{d_j \subset d_j'\} \\
&\{C \triangleright \text{width}(t_j([a_i \mapsto e_i])) = \text{width}(t_j')\} \\
\frac{\Delta; \Gamma; C \vdash P}{\Delta; \Gamma; C \vdash m(e_i) \langle i_j \rangle} \quad \text{(T-Mod)}
\end{align*}
\]

\[
\begin{align*}
&\text{out} \in d_1 \quad \text{in} \in d_2 \\
&\{\Delta; \Gamma; C \vdash l : d_1 t_1\} \\
&\{C \triangleright \text{width}(t_1) = i\} \\
\frac{\Delta; \Gamma; C \vdash \text{assign } l \langle 0[1] \rangle}{\Delta; \Gamma; C \vdash \text{assign } l \langle 1 \rangle} \quad \text{(T-Assign1)}
\end{align*}
\]

\[
\begin{align*}
&e \notin \{0[1]\} \\
&\text{out} \in d_1 \quad \text{in} \in d_2 \\
&\{\Gamma; C \vdash_1 l : d_1 t_1\} \\
&\{\Gamma; C \vdash e : d_2 t_2\} \\
&\{C \triangleright \text{width}(t_1) = \text{width}(t_2)\} \\
\frac{\Delta; \Gamma; C \vdash \text{assign } l \langle e \rangle}{\Delta; \Gamma; C \vdash \text{assign } l \langle c \rangle} \quad \text{(T-Assign2)}
\end{align*}
\]

\[
\begin{align*}
&\{\Delta; \Gamma; C; e \vdash P_j\} \\
&\frac{\Delta; \Gamma; C \vdash \text{if } e \text{ then } (P_j) \text{ else } (P_j')}{\Delta; \Gamma; C \vdash \text{if } e \text{ then } (P_j) \text{ else } (P_j')} \quad \text{(T-If)}
\end{align*}
\]

\[
\begin{align*}
&\{\Delta; \Gamma; y : \text{int} \vdash e_1, e_2 : \text{int}\} \\
&\frac{C \triangleright e_3 > 0}{\Delta; \Gamma; C \vdash \text{for}(y = e_1, y < e_2, y = y + e_3) (P_j)} \quad \text{(T-For)}
\end{align*}
\]

**Figure 3.3** Type System For Parallel Statements

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 the newly created module need to be substituted into to reflect the values of the parameters that were used in the instantiation. We display the substitution definition with the required extensions in Figure 3.5 and the definition of the operation semantics in Figure 3.6. The potential preprocessing errors are unchanged from those defined in Appendix C except for minor syntactic adjustments.
### 3.6 Technical Results

In this section we restate the main theorems from the previous chapter to confirm that they still hold. The complete proofs of these theorems are in Appendix D.

#### 3.6.1 Type Preservation

A well-typed description produces a well-typed description. Formally:
\[ P[x \mapsto v] \]
\[ m(e_1)(l_i)[x \mapsto v] = m(e_1[x \mapsto v])(l_i[x \mapsto v]) \]
\[ (\text{assign } l \cdot e)[x \mapsto v] = \text{assign } l[x \mapsto v] e[x \mapsto v] \]
\[ (\text{if } e \text{ then } P_1 \text{ else } P_2)[x \mapsto v] = \text{if } e[x \mapsto v] \text{ then } P_1[x \mapsto v] \text{ else } P_2[x \mapsto v] \]
\[ (\text{for}(y = e_1; y < e_2; y = y + e_3)(P_1))[x \mapsto v] = \text{for}(y = e_1[x \mapsto v]; y < e_2[x \mapsto v]; y = y + e_3[x \mapsto v])(P_1[x \mapsto v]) \]

\[ l[x \mapsto v] \]
\[ \text{See } c[x \mapsto v] \]

\[ c[x \mapsto v] \]
\[ s[x \mapsto v] = s \]
\[ s[e][x \mapsto v] = s[e[x \mapsto v]] \]
\[ s[e_1; e_2][x \mapsto v] = s[e_1[x \mapsto v]; e_2[x \mapsto v]] \]
\[ x[x \mapsto v] = v \]
\[ y[x \mapsto v] = y \text{ if } y \neq x \]
\[ t'[x \mapsto v] = t' \]
\[ f(e_i)[x \mapsto v] = f(e_i[x \mapsto v]) \]

\[ e[x \mapsto v] \]
\[ (e_1 \text{ } R \text{ } e_2)[x \mapsto v] = e_1[x \mapsto v] \text{ } R \text{ } e_2[x \mapsto v] \]
\[ (\neg e)[x \mapsto v] = \neg(e[x \mapsto v]) \]

\[ \Gamma[x \mapsto v] \]
\[ [][x \mapsto v] = [] \]
\[ (s : d \text{ :: } \Gamma)[x \mapsto v] = s : d \text{ :: } t[x \mapsto v] :: \Gamma[x \mapsto v] \]
\[ (x : d \text{ :: } \Gamma)[x \mapsto v] = x : d \text{ :: } t[x \mapsto v] :: \Gamma[x \mapsto v] \]

\[ t[x \mapsto v] \]
\[ \text{int}[x \mapsto v] = \text{int} \]
\[ \text{wire}[x \mapsto v] = \text{wire} \]
\[ t(e_1, e_2)[x \mapsto v] = t[x \mapsto v](e_1[x \mapsto v], e_2[x \mapsto v]) \]

Figure 3.5 Substitution

**Theorem 4 (Type Preservation).** If \( \vdash p \) and \( p \Downarrow p' \) then \( \vdash p' \)

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

\[ \square \]

### 3.6.2 Type Safety

We also want to prove type safety:
Theorem 5 (Type Safety). If ⊢ p and $p \xrightarrow{1} p'$ then $p' \neq \text{err}$

Sketch. This result follows directly from Theorem 4 since no typing rules will consider err well-typed.

3.6.3 Preprocessing Soundness

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

$$\begin{align*}
\text{Preprocessed Term } & \bar{X} = \\
\{ u \mid u \in X, \land Y \in \text{subterms}(u) \Rightarrow \\
( & (Y = (e_i)_{i \in I}(d_j t_j a_j)_{j \in J} \text{ is } (t_k y_k)_{k \in K} (P_r)_{r \in R} \Rightarrow I = \emptyset) \\
\land & (Y = m (e_i)_{i \in I} (t_j)_{j \in J} \Rightarrow I = \emptyset) \\
\land & (Y \neq \text{if } e \text{ then } (P_i)_{i \in I} \text{ else } (P_j)_{j \in J}) \\
\land & (Y \neq \text{for}(y = e; y < e; y = y + e) (P_i)_{i \in I})\}\end{align*}$$

Theorem 6 (Preprocessing Soundness). If $p \xrightarrow{1} p'$ then $p' \in \bar{p}$

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

3.7 Related Work

To the best of our knowledge, no one has considered Verilog static checking 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 include Polyspace Ada/C/C++ Verifier [57] and C Global Surveyor [58]. 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 is only required 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 [59] is such an example. It does lightweight analysis but it is neither sound nor complete. ESPX [60] 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 SAILInfer 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.

Our type checker does not need to do inter-modular analysis and it does not require the programmer to put any additional annotations. The reason behind that 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 don't 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, which 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 [61], 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 type 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.

3.8 Implementation

In addition to checking for correct level separation and synthesizability, VPP’s type checker implements all the ideas discussed in this chapter. VPP makes use of Yices, a state of the art SMT solver [62]. Since VPP is developed in OCaml, we developed an OCaml library to communicate with the C APIs provided by Yices.

Any efficient SMT solver supporting integer linear arithmetic and uninterpreted function theories can be used. The integer linear arithmetic theory is used in various places, especially when translating Verilog expressions into satisfiability expressions.

The uninterpreted functions are very useful to define non-linear functions like min, max, and power. For example here is how we define min in Yices:

```ocaml
(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.
3.8.1 Limitations

Relying on an SMT solver to decide if a description is well-typed imposes certain restrictions on our type checking method due to the limitations in the solver. For instance, SMT solvers cannot handle non-linearities. Additionally, the language accepted by the SMT solver restricts the constraints that can be sent to it. For instance converting Verilog expressions to satisfiability expressions is not always straightforward even using uninterpreted functions. When dealing with a description whose consistency cannot be verified statically due to Yices limitations we consider it as ill-typed and reject it.

3.8.2 Performance Evaluation

We wrote some circuit family descriptions 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 A.

Table 3.1 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 ex-
ample. 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 (SMT). These experiments were conducted on the same machine described in the previous chapter: A 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 spends 126 ms solving satisfiability problems despite not being the longest description because of its extensive use of two-dimensional arrays.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Code Size (lines)</th>
<th>VPP’s Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Code Size (msec)</td>
<td>P (msec)</td>
</tr>
<tr>
<td>Inverter</td>
<td>10</td>
<td>0.32</td>
</tr>
<tr>
<td>Counter</td>
<td>20</td>
<td>0.41</td>
</tr>
<tr>
<td>Parity</td>
<td>13</td>
<td>0.34</td>
</tr>
<tr>
<td>Ripple Adder</td>
<td>27</td>
<td>0.51</td>
</tr>
<tr>
<td>Carry Select</td>
<td>59</td>
<td>0.77</td>
</tr>
<tr>
<td>Decoder</td>
<td>22</td>
<td>0.40</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>21</td>
<td>0.46</td>
</tr>
<tr>
<td>Multiplier</td>
<td>39</td>
<td>0.66</td>
</tr>
</tbody>
</table>
3.9 Summary

This chapter showed that using an advanced type system, we can statically check circuit families for hidden sources of incorrectness or inefficiency. In our opinion this kind of checking is essential if we expect designers to trust generic constructs and start writing meaningful descriptions of circuit families. The next chapter shows that by further extending the type system it can be used to provide guarantees on the number of gates required to physically realize a circuit family. This kind of guarantee is equally important in helping the designers quickly see the effect of their design decisions on the physical properties of their creations.
\[
\begin{align*}
&D \vdash (D_r \rightarrow (D_r) \vdash b \vdash_{b_1} b', (D_r) \vdash_{b_1} b') (D_r) \vdash_{b_1} (D_r) \vdash (module \ b \in (D_r)), (D_r) \vdash (module \ b \in (D_r)) (E-Prog) \\
&D \vdash (D_r) \vdash (D_r \vdash_{b_1} b') \vdash (D_r) \vdash (module \ b \in (D_r)) (E-Body) \\
&D \vdash (D_r) \vdash (assign \ l \vdash_{b_1} l', e \vdash_{b_1} e', (D_r)) (E-Assign) \\
&D \vdash (D_r) \vdash (assign \ l \vdash_{b_1} l', e \vdash_{b_1} e', (D_r)) (E-Mod) \\
&D \vdash (D_r) \vdash (assign \ l \vdash_{b_1} l', e \vdash_{b_1} e', (D_r)) (E-ForTrue) \\
&D \vdash (D_r) \vdash (assign \ l \vdash_{b_1} l', e \vdash_{b_1} e', (D_r)) (E-ForFalse) \\
\end{align*}
\]

Figure 3.6 Operational Semantics
Chapter 4
Parametric Gate Count Analysis

In addition to design complexity, circuit designers face challenges in terms of tool chain predictability. New approaches to design are needed to overcome these challenges. A promising approach is to combine powerful abstraction mechanisms for managing design complexity with powerful static analyses that can penetrate the complexity. As a step in this direction, we show the feasibility of pre-elaboration evaluation of gate counts in circuit families.

4.1 Design Space Exploration

This chapter shows that the type of static analysis used in the previous chapters can be pushed one step further to derive closed-form expressions for gate count in generic circuits. To this end, we present a method to compute the gate count of a parameterized module as a function of its parameter values. Gate count facilitates rapid design exploration as follows:

- It is a key factor affecting the area and power consumption of a circuit.
- It can be used to compare various implementations of the same circuit.
- Gate count can be applied directly to FPGA-based design [63].
Current methods and tools provide actual gate count after elaboration and synthesis for a particular target technology. Static parameterized evaluation of gate count has the following advantages:

- **Design Space Exploration:** For a parameterized module, a numeric gate count obtained for a particular set of parameter values is less informative than a parametric gate count function. A parametric expression allows the designer to immediately see and understand the effect of changing the parameter values on gate counts.

- **Speed:** Elaboration and synthesis are often very time consuming processes. If various implementations of the same functional unit are to be implemented and compared (e.g. to optimize a particular design), it is much more efficient to compare their relevant properties statically than having to go through the lengthy and complicated synthesis process.

- **Modularity:** After elaboration and synthesis, only a global gate count for the whole design is obtained. It is hard to estimate the effect of modifying or replacing a particular module on the obtained global design.

### 4.2 Contributions

We show that gate count can be mechanically computed for generic descriptions. This is achieved as follows:
• We develop a method to statically analyze a parameterized structural hardware description allowing us to compute its gate count as a function of its input parameters (Section 4.3).

• The method is formalized and proven correct (Section 4.4). The formalization is based on an extension to the type system of Featherweight Verilog (FV) where the type signature of each module is augmented with a parametric gate count expression.

• We explain how to use the presented approach to statically analyze a parameterized structural hardware description for other resources (Section 4.5)

Upon request, VPP’s type checker can provide a parametric gate count for each module analysed (Section 4.7)

4.3 Approach

The method can be explained by considering the ripple-carry adder family repeated here for convenience.

```verilog
module ripple_adder(s,cout,a,b,cin);
    parameter N=4;
    input [N-1:0] a,b;     input cin;
    output [N-1:0] s;      output cout;
    wire [N:0] c;         genvar i;

assign c[0] = cin;
generate
    for(i=0; i<N; i=i+1)
        full_adder fa (s[i],c[i+1],a[i],b[i],c[i]);
endgenerate
```
assign cout = c[N];
endmodule

Clearly the gate count of this module is going to be a function of \( N \), but first we need to know the number of gates required by a single full adder. A reasonable structural definition for a full adder might look as follows:

```verilog
module full_adder (sum,cout,a,b,cin);
    input  a,b,cin;
    output sum,cout;
    wire w1,w2,w3;

    xor(w1,a,b);
    xor(sum,w1,cin);
    and(w2,cin,w1);
    and(w3,a,b);
    or(cout,w2,w3);
endmodule
```

Given this definition it can be seen that a full adder requires 5 gates and since the \( N \)-bit ripple adder is composed of \( N \) full adders (and nothing more), then the parametric gate count of the ripple adder module is expected to be \( 5N \). This parametric value is much more informative then a gate count of 20 based on the default parameter value.

We use a type based approach to provide a sound gate count analysis. The main idea is that our type system is designed to infer the gate count of each module as part of its type. In the simplest cases, this amounts to counting the gates used in the module description. More work is done when hierarchical descriptions or generative constructs are used. When the module is not parameterized, the computed gate count
is just an integer (as in the `full_adder` module) otherwise it can be a function of the input parameters (as in the `ripple_adder` module).

### 4.4 Formal Account

We formalize our method as an addition to the Featherweight Verilog (FV) calculus. Both the syntax and operational semantics remain unchanged from those defined in the previous chapter. In this chapter we extend FV's type system to statically infer parametric gate count expressions. In the next section we explain in more detail how gate count is computed for each construct.

#### 4.4.1 Additions to Type System

In this section we explain how FV's type system is used to infer gate count expressions. We focus only on the extensions made to FV's type system to allow for gate count analysis. These extensions are highlighted in gray.

We first start by defining the syntax of gate count expressions:

\[
\text{Count Expression } \quad g \in \mathbb{G} \quad ::= \quad V \mid z \mid h(g_i)^{\text{ef}} \mid 2^g \mid g\{g \mid \sum_{i=g}^{g} g}
\]

A gate count expression \( g \) can either be an integer \( V \), a variable \( z \), an operator application \( h(g_i)^{\text{ef}} \), an exponential of base 2, a conditional expression, or a summation. Binary operators include arithmetic, comparison, and logical operators. An FV expression whose operands are either constants or parameters can be translated to
an equivalent gate count expression via a straightforward translation function that we denote by $[.]_g$.

Additionally we use the following auxiliary notions:

\[
\begin{align*}
\text{Module Type} & \quad M := (x_i)^{i \in I} (d_j t_j)^{j \in I} [g] \\
\text{Operator Signatures} & \quad \Phi \in \Pi \times \mathbb{O} \times \mathbb{N} \times \mathbb{T}^2 \rightarrow \mathbb{T} \times \mathbb{G}
\end{align*}
\]

A module type or signature is now a sequence of parameters, a sequence of directions and types for ports, and a gate count expression $g$. An operator signature is a function that given a number of operands, an operator, a type checking level (FV is a two-level language), and a sequence of types of all the operands, returns the type of the result in addition to a gate count required to produce that output. It is important to note that FV’s type system uses dependent types, allowing it to distinguish wires of different widths at the type level. This allows the operator signature to return different gate counts as functions of the wire widths used. For example if two wires of width 8 are xored with the bitwise $\wedge$ operator, the gate count expression returned by $\Phi$ will be 8 based on the dependent types of the operands.

As shown in Figures 4.1 and 4.2, FV’s type system assigns each module a type signature containing a gate count expression. Since a module is mainly composed of a set of parallel statements, the total number of gates is simply the sum of the gate counts for each of these statements (T-Body). Gate count for parallel statements is computed as follows:

- **Assignment Statement**: The right hand side of an assignment statement
is an expression that might require some gates (T-Assign2). This of course depends on the expression itself. If the expression is an LHS value, a variable, or an integer then the gate count is 0 (T-Id, T-Idx, T-Rg, T-Par, T-Int). The last case is when the expression is an operator application in which case we compute the gate count for each of the operands and add the sum to the gate count returned by looking up the operator signature (T-Op).

- **Module Instantiation:** When a module is instantiated, its type signature (presumably inferred earlier) includes the corresponding gate count expression. Care must be taken if the instantiated module is parameterized, in which case we substitute the parameters in the gate count expressions with the actual values used at instantiation time. Note that since there are no default parameter values in FV, it is necessary to provide parameter values when instantiating a parameterized module (T-Mod).

- **Generative Conditional:** First, we compute \( \sum_i g_i \) and \( \sum_j g_j \) which are the total gate count expressions for the consequent and alternative statements respectively and then we combine these using a ternary conditional operator.

- **Generative Loop:** First, we compute \( \sum_i g_i \) as the total gate count expression for the loop body. Because this expression might depend on the loop index, the actual number of gates is obtained by its summation for all the values taken by
the loop index. If the loop header is for($y = e_1; y < e_2; y = y + e_3$), then the actual gate count is equal to sumstep($\sum_{i} g_i[y, [e_1], [e_2], [e_3]]$) where sumstep is defined as follows:

$$\text{sumstep}(g, y, g_1, g_2, g_3) = \sum_{y=g_1/g_3}^{(g_2-1)/g_3} g\left[y \mapsto y * g_3\right]$$

The obtained summation is standard in the sense that $y$ is incremented by 1 in each of the terms of its expansion. The reason $y$ is substituted with $y * g_3$ inside the gate count expression is to account for the fact that the original loop had a increment that is potentially unequal to 1. The summation bounds are divided by $g_3$ for the same reason. Unfortunately, this division means that the summation bounds are not guaranteed to be integers. In many cases the resulting summation can be converted into a closed form using the well-known mathematical properties of summations as shown in the next section.

The total gate count of a whole circuit is simply the gate count of its top level module (T-Prog).

4.4.2 Summation Closed Forms

As stated above, in many cases, the obtained summation can be converted into a closed form. In general the obtained summation is of the form $\sum_{y=g_1}^{g_2} g$ where both $g_1$ and $g_2$ can be any arbitrary gate count expressions not necessarily representing integer values. To get a closed form, we use the set of identities shown in Figure 4.3.
\[ \Delta; \Gamma; C \vdash P : g \]

\[
\begin{align*}
\{ \Gamma; C \vdash^0 e_i : \{ \text{in} \} \text{ int } 0 \} & \quad \Delta(m) = (x_i) (d_j t_j) \ g \\
\{ \Gamma; C \vdash l_i : d_j t_j \ 0 \} & \quad (d_j \in d'_j) \\
\{ C \triangleright width(t_j) \{ [x_i \mapsto e_i] \} \} = width(t'_j) & \\
\end{align*}
\]

\[ (\text{T-Mod}) \]

\[
\begin{align*}
\Delta; \Gamma; C \vdash m([n_i]) (l_j) : g([x_i \mapsto [e_i]] & \\
\end{align*}
\]

\[ (\text{T-Mod}) \]

\[
\begin{align*}
e \neq (0(1))^+ & \\
\text{out} \in d_1 & \quad \text{in} \in d_2 \\
\Gamma; C \vdash l : d_i t_i \ 0 & \\
\Gamma; C \vdash e : d_i t_i \ g & \\
\end{align*}
\]

\[ (\text{T-Assign1}) \]

\[
\begin{align*}
\Delta; \Gamma; C \vdash \text{assign } l e : g & \\
\end{align*}
\]

\[ (\text{T-Assign2}) \]

\[
\begin{align*}
\Delta; \Gamma; C \vdash e : \{ \text{in} \} \text{ int } 0 & \\
\{ \Delta; \Gamma; C, e \vdash P_i : g_i \} & \\
\Delta; \Gamma; C, e \vdash \text{If } e \text{ then } P_i \text{ else } P_j : ([e_i] \sum_{i} g_i : \sum_{j} g_j) & \\
\end{align*}
\]

\[ (\text{T-If}) \]

\[
\begin{align*}
\Gamma, y : \{ \text{in} \} \text{ int } C \vdash^0 e_3 : \{ \text{in} \} \text{ int } 0 & \\
\Gamma; C \vdash e_1, e_2 : \{ \text{in} \} \text{ int } 0 & \\
\{ C \triangleright e_1 > 0 & \\
\Delta; \Gamma; C \vdash \text{for} (y = e_1 ; y < e_2 ; y = y + e_3) (P_i) : \text{sumstep} \left( \sum_{i} g_i, y, [e_1]_x, [e_2]_y, [e_3]_z \right) & \\
\end{align*}
\]

\[ (\text{T-For}) \]

**Figure 4.1** Type System for Parallel Statements

### 4.4.3 Correctness

Now we are ready to establish a soundness theorem stating that the actual gate count after elaboration is equal to our statically computed gate count. To do so, we refer to FV’s elaboration semantics as defined in the previous chapter. In the following, we use the term exact gate count for a program \( p \). The exact gate count is only defined on fully elaborated descriptions and as such is always an integral value.

First, we establish that the gate count for an elaborated program is exact:

**Lemma 1.** If \( p \xrightarrow{1} p' \) and \( \vdash p' : g \) then \( g \) is the exact gate count for \( p' \)

**Proof.** Using the preprocessing soundness theorem established for FV 6, we know
that \( p' \) is free from all generative constructs. It does not include any parameterized modules, conditionals, or loops. This means that \( g \) is merely the sum of all gates composing \( p' \) and is therefore an exact count.

Next, we show that gate count is preserved by elaboration:

**Lemma 2.** If \( \vdash p : g \) and \( p \vdash \xrightarrow{1} p' \) and \( \vdash p' : g' \) then \( g = g' \)

**Proof.** The proof proceeds by induction on the derivation of the second judgment.
\[ \sum_{y=g_1}^{g_2} g = [(g_2 - g_1 + 1)]g \quad (y \notin g) \]
\[ \sum_{y=g_1}^{g_2} y = \frac{(g_2 - g_1 + 1)((g_2 + g_1) + 1)}{2} \quad g_1 \in \mathbb{N} \]
\[ \sum_{y=1}^{g_2} y^2 = \frac{g_2[(g_2 + 1)(2g_2 + 1)]}{6} \]
\[ \sum_{y=1}^{g_2} y^3 = \left( \sum_{y=1}^{g_2} y \right)^2 \]
\[ \sum_{y=g_1}^{g_2} (yg_4) = g_3 \sum_{y=g_1}^{g_2} g_4 \quad y \notin g_3 \]
\[ \sum_{y=g_1}^{g_2} (g_3 + g_4) = \sum_{y=g_1}^{g_2} g_3 + \sum_{y=g_1}^{g_2} g_4 \]
\[ \sum_{y=g_1}^{g_2} (g_3 \cdot g_4 : g_5) = g_3 \sum_{y=g_1}^{g_2} g_4 : \sum_{y=g_1}^{g_2} g_5 \quad y \notin g_3 \]
\[ \sum_{y=g_1}^{g_2} ((y <= g_3) \cdot g_4 : g_5) = \sum_{y=g_1}^{g_2} g_4 + \sum_{y=g_1}^{g_2} g_5 \]

Figure 4.3 Summation Identities

It is important to note that in FV, the top level module of a program \( p \) cannot be parameterized which guarantees that \( g \) in lemma 2 is always an integral value just like \( g' \).

Now, it is immediate that the gate count given by the pre-elaboration analysis is sound:

**Theorem 7.** If \( \vdash p : g \) and \( p \xrightarrow{1} p' \) then \( g \) is the exact gate count for \( p' \)

**Proof.** This theorem follows directly from combining the previous two lemmas. \( \square \)
4.5 Analysis for Other Resources

The same method can be extended to statically and parametrically compute other kinds of resources consumed by the circuit. In this study we have shown how to count gates collectively, but counting a particular kind of gates (e.g. 2-input and gates) can clearly be done in exactly the same way. Similarly counting the number of wires required by a circuit is a straightforward application. Approximate estimates for more complex resources like area or power consumption can be computed using the same approach provided that the area and power consumed by each primitive component is known. Once known the same type system can be used to compute the required parametric estimate by replacing the operator signature $\Phi$ with one reflecting the cost of each primitive operation.

4.6 Related Work

To the best of our knowledge, no one has considered automatic parametric gate count estimation before. Nemani and Najm [64] suggested a method to statically estimate the number of gates required for an optimal implementation given boolean equations describing a circuit. Their approach relies on heuristics and is concerned with the low level details of synthesis optimization. More importantly it does not support generic descriptions and cannot generate parametric gate count estimates. Brandoese et al. [65] presented an area estimation methodology for FPGA designs at the SystemC level. Our technique can be combined with theirs. From a type
theoretic point of view, our approach is closer to the approach used by Taha, Ellner and Xi [46] as they relied on statically typed two-level languages to generate heap bounded implementation of sorting programs. However their method is not directly applicable to hardware description languages and they do not attempt to generate parametric estimates of the heap resources required by the generated program.

4.7 Implementation

As described earlier, VPP first type checks the input description verifying that it is synthesizable and inconsistency free. Additionally, VPP's type checker is enriched with parametric gate counts for each module based on the rules defined above. For the top level module, our tool always generates an integer gate count in addition to the parametric one. This numeric value corresponds to the gate count of the whole circuit and is directly obtainable if the top level module is non-parameterized; otherwise it is obtained by evaluating the parametric gate count using the default parameter values.

If the input description is proved well-typed, it is then elaborated producing a Verilog description that is free of generative constructs. The gate count in the resulting description is trivial since all the gates are explicitly shown.

To compute the number of gates used by a certain module, it is necessary to know the gate count of all the modules composing it. As shown in the introductory example, we needed the definition of the full_adder module in order to compute the gate count of the ripple_adder module. When a designer is only concerned with the gate count
of a particular module, forcing him to provide the complete implementation of all the composing modules (and the ones composing them and so on) is impractical. In fact it is not necessary: the only reason we need these definitions is to statically analyze them, and infer their gate counts. Alternatively the designer might opt to specify the gate count of composing modules by only providing their type signature including the gate count. So instead of providing the complete full_adder module definition, our implementation would also accept the following declaration:

```plaintext
assume full_adder (output wire sum, output wire cout,
    input wire a, input wire b,
    input wire cin) 5;
```

The value 5 at the end represents the gate count of the assumed module. If the assumed module is parameterized then the integral gate count can be replaced by an expression.

4.7.1 Experimental Evaluation

To test our tool, we applied it to the same circuit families we used in the previous chapter. The results obtained for each of them are shown in table 4.1.

The actual post-elaboration gate count when instantiating these modules with various parameter values always agreed with the statically computed value. It is important to note that all these examples contain generative loops resulting in summations and that for all these examples the summation was converted into a closed form. The following family shows an example where the gate count is a conditional
Table 4.1  Parametric Gate Count For Circuit Families

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Parametric Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inverter</td>
<td>$N$</td>
</tr>
<tr>
<td>Counter</td>
<td>$10N$</td>
</tr>
<tr>
<td>Parity</td>
<td>$N - 1$</td>
</tr>
<tr>
<td>Ripple Adder</td>
<td>$5N$</td>
</tr>
<tr>
<td>Carry Select</td>
<td>$14N + 4$</td>
</tr>
<tr>
<td>Decoder</td>
<td>$2^N + N$</td>
</tr>
<tr>
<td>Mux</td>
<td>$2^{N+1} + N + 1$</td>
</tr>
</tbody>
</table>

expression:

module addative_adder (sum, cout, a, b, cin);
  parameter N = 8;
  input [N-1 : 0] a, b;
  input cin;
  output [N : 0] sum;
  output cout;
  generate
    if(N<=8)
      cs_adder #(N) cladder (sum, cout, a, b, cin);
    else
      ripple_adder #(N) radder (sum, cout, a, b, cin);
  endgenerate
endmodule

This example describes an $N$-bit adder block that internally is either a ripple or a carry select adder depending on the parameter value used to instantiate it. The value 8 assigned to $N$ is only a default value. Consequently, the number of gates used is either the number of gates needed by a ripple adder ($5N$) or the number of gates used by the carry select adder ($14N+4$). The inferred gate count is $(N <= 8)?14N+4 : 5N$. 
4.8 Summary

This chapter argued that dependent types can be used to get a quick parametric gate count for circuit families and that the same technique can potentially be used to provide estimates for other resources. This concludes the set of static analyses that we propose to make the existing generative constructs of Verilog usable in a practical setting. The next chapter proposes adding new generative constructs as aimed at making Verilog more expressive. It also deals with the implications of these additions on the static analyses we already presented.
Chapter 5
Increasing Verilog’s Generative Power

Despite its wide use in industry, Verilog provides limited support for capturing parametric circuit descriptions. We have seen earlier that generative constructs offer a concise way of describing circuit families; however, these constructs are limited to loops and conditionals. Additionally, module parameters are restricted to integer values.

In this chapter, we address these expressivity limitations, identifying the key extensions needed to overcome them, and showing that our techniques for statically checking parameterized Verilog modules can be modified to support these significant extensions.

5.1 Contributions

To increase Verilog’s generative power we:

- List the main missing features that limit Verilog’s expressivity (Section 5.2).

- Describe the proposed extended Verilog syntax by means of illustrative examples (Section 5.3).

- Informally explain the elaboration semantics of the proposed constructs (Section 5.4).
• Informally describe how to extend the various static analyses to support the new constructs (Section 5.5).

• Provide an implementation supporting the new constructs (Section 5.6).

5.2 Missing Language Features

Verilog’s support for generative constructs provides a natural way to capture the design of circuits with linear structures. While many important circuits like adders, multipliers, and counters fit this pattern, there are other equally important circuits that have so far required ad hoc program generation techniques to describe. Examples of such circuit patterns include tree shaped circuits [66] and butterfly circuits [67]. Additionally, even when they only feature linear structures, generic composition patterns like the common ripple pattern shown in Figure 5.1, cannot be expressed in Verilog. These expressivity restrictions do not exist in functional hardware description languages like Lava [1]. For example, the ripple pattern can be described in Lava as follows:

\[
\text{ripple circ } (\text{cin}, [1]) = ([1], \text{cin})
\]

\[
\text{ripple circ } (\text{cin}, a:as) = (b:bs, cout)
\text{ where}
\]

\[
(b, \text{carry}) = \text{circ } (\text{cin}, a)
\]

\[
(bs, cout) = \text{ripple circ } (\text{carry}, as)
\]

Describing such patterns requires the use of 1) recursion, 2) polymorphism, and 3) the ability to parametrize a circuit by other circuits. These points were made in
Figure 5.1  The Ripple Pattern Connecting n Instances of circ (adapted from [1])

early works by Sheeran et al on Lava [1], and by O’Leary et al on HML [68]. These ideas have also been incorporated into systems such as BlueSpec [47].

Our goal is to re-introduce these three features into Verilog as extensions that can be implemented through a preprocessor, that is, as syntactic sugar. In addition, and most importantly, we demonstrate that powerful static analyses are still possible with these extensions.

Achieving these goals requires: 1) extending the syntax in a manner that is compatible with standard Verilog, 2) defining the semantics for the new extensions by a translation into Verilog, and 3) defining the static analyses at the level of this extended source language.

5.3 Proposed Syntax

For syntax, we propose a syntax which would allow expressing the above Lava example as follows:

```verilog
module ripple
  (output 't1 [N-1:0] b, output 't2 cout,
   input 't3 [N-1:0] a, input 't2 cin);
```
parameter \( N=8 \) where \( N \geq 1 \);
parameter circ (output 't1 y, output 't2 co,
           input 't3 x, input 't2 cin);

't2       carry;
circ c (b[0],carry,a[0],cin);

if (N == 1)
  assign cout = carry;
else begin
  ripple ##(circ) #(N-1) ra (b[N-1:1],cout,a[N-1:1],carry);
end
endmodule

The above generic module recursively describes the ripple pattern. It is parametrized
by an integer parameter \( N \) and by the module circ whose full signature is specified.
It also allows the ports a and b to be arrays of arbitrary types. Being parameter-
ized by a module, the ripple pattern is a higher order module. When instantiating
a higher order module, module parameters must be passed in before passing in the
integer parameter values (if any). To easily differentiate between module and integer
parameters, the former is preceded by ## as opposed to #.

Although fairly concise, the Verilog description is still significantly longer than its
Lava counterpart primarily due to its explicit type declarations (as opposed to Lava’s
inferred types). This price cannot be avoided since we aim to provide extensions that
fit naturally with the rest of Verilog, and that existing Verilog users can easily adopt.

In addition, the ripple pattern example shown above features a \texttt{where} clause in the
parameter declaration statement. This is also a Verilog extension that we propose.
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.

The last extension we propose is to fully support multidimensional arrays. The latest Verilog standard supports multidimensional arrays but not for module ports. A port can either be a single bit or a simple array (a bitvector). We find this restriction quite limiting and we propose removing it. This will allow us for example to define a generic bus multiplexer as follows:

```verilog
class gen_bus_mux(mux_out, mux_in, sel);
  parameter M=2 where M >= 1;
  parameter W=8;
  input [W-1:0] mux_in [2**M-1:0];
  input [M-1:0] sel;
  output [W-1:0] mux_out;
  genvar i,j;
  wire [2**M-1:0] decsel;
  wire [2**M-1:0] p [W-1:0];
  decoder #M decl (decsel, sel);
  for(i=0;i<2**M;i=i+1)
    for(j=0;j<W;j=j+1)
      and (p[j][i], decesl[i], mux_in[i][j]);
  for(i=0;i<W;i=i+1)
    assign mux_out[i] = |p[i]; //unary reduce operator
endmodule
```

This module uses two parameters `M` and `W` to specify the bus width and the number of selection lines respectively. The number of selection lines is restricted to be at least
one. This module has 3 ports, two of which are simple arrays (\texttt{mux\_out} and \texttt{sel}). The third port (\texttt{mux\_in}), however, needs to be an array of buses which means it must be an array of array. We use Verilog’s standard notation in specifying multidimensional arrays to specify that \texttt{mux\_in} is a multidimensional array (an array of bitvectors to use Verilog’s terminology).

To summarize, we propose the five following extensions:

- Recursive modules
- Higher order modules
- Parametric polymorphism
- Parameter constraints
- Multidimensional ports

In the next sections we consider the implications of these extensions on the elaboration semantics and the supported static analyses.

5.4 Semantics of Proposed Extensions

To ensure backward compatibility, any extensions we make to Verilog need to be fully expanded away during elaboration. We assume that we only elaborate well-typed programs since type checking is performed before elaboration. Here is what needs to be done for each of the proposed additions:
• **Recursive modules**: Surprisingly, nothing needs to be changed in the elaboration semantics to support recursive modules. A well-typed recursive module never instantiates itself with the same parameter values it received. This is guaranteed by the termination analysis we present in the next section. This means that standard elaboration will successfully unroll recursion completely. To see why, consider an N-bit ripple adder defined recursively and then instantiated to create a 16-bit ripple adder. Elaborating this instantiation causes the creation of a 16-bit ripple adder module composed of a full adder and an instantiation of a 15-bit ripple adder. This instantiation in turn creates a 15-bit ripple adder module using a 14-bit ripple adder, and so on until the base case (a 1-bit ripple adder) is reached. After elaboration none of the resulting modules is recursive which means that recursive modules are expanded away on their own without any modifications to our preprocessing rules.

• **Higher order modules**: Elaborating higher order modules is very similar to elaborating parameterized modules. When instantiated, a higher order module must be given the name of the module(s) to use internally. This in turn creates a specialized (first order) module where the names of the modules being instantiated are substituted by the given module names.

• **Parametric polymorphism**: A polymorphic module can only be instantiated
with concrete types. The polymorphic types are replaced by their corresponding concrete types creating a specialized (non polymorphic) module. The concrete types to use are determined during the type checking phase as explained in the next section.

- **Parameter constraints**: Elaboration soundness implies that during elaboration all parameter declarations and their associated where clauses are dropped.

- **Multidimensional ports**: Multidimensional ports are replaced by a series of single dimensional ports during elaboration. This can only be done when a module is instantiated since the number of generated ports might depend on a parameter value as in the generic bus multiplexer example. This change is reflected in the module ports declaration and when the module is instantiated.

5.5 Static Analyses

Adding new constructs to a language is almost guaranteed to increase the complexity of static analyses defined for that language. This is particularly true of generative constructs. Here is what needs to be done for each of the proposed extensions:

- **Recursive modules**: Recursive modules are the most challenging extension to handle. There are 3 modifications to type checking that are required: 1) Type checking needs to be done in two passes. The first one to infer all the module types and the second one to actually type the module definitions. 2) Static
resource estimation needs a recurrence relation solver because the amount of resources required by a module may potentially be a recurrence. We suggest using PURRS [69] or PUBS [70] to obtain a closed form estimate or content ourselves with recurrences as gate count estimates. 3) Type checking now requires a more elaborate termination analysis due to the possibility of preprocessing divergence. We present the analysis we use in Section 5.5.1.

- **Higher order modules**: To support higher order modules, the type checker needs to check that in the definition of the higher order module each instantiation using a module parameter has a type compatible with its declaration. Additionally, when a higher order module is instantiated, the type checker needs to verify that the module passed as a parameter is compatible with the expected type.

- **Parametric polymorphism**: To verify that a polymorphic module is instantiated correctly, the type checker needs to infer the concrete types corresponding to the polymorphic types. We use unification [71] as suggested by Hindley-Milner type inference [72,73].

- **Parameter constraints**: Supporting where clauses is a straightforward extension. First, when type checking the definition of the parameterized module with a restricted parameter, we consider the parameter constraint to be a given that
we can use to prove other consistency constraints. Second, the type checker verifies that the default parameter values satisfy the constraints imposed on them. Similarly, at the instantiation site, the type checker verifies that the passed parameter values also satisfy the same constraints. Parameter constraints are also used for termination analysis as described in the next section.

- **Multidimensional ports**: Nothing special needs to be done to support multidimensional ports during type checking.

### 5.5.1 Termination Analysis

Termination analysis is an active research area [74–76]. In our proposed version of Verilog, there are only two potential reasons for elaboration divergence: loops and recursive calls. Since we restrict loops to a very simple form, our type checker is already able to verify their termination by using an SMT solver to prove that the loop index increases with each iteration. Since we allow mutual recursion, proving termination of recursive calls requires more work despite the restricted nature of Verilog’s modules which only allow integer parameters. We propose an abstract interpretation [77,78] of the description using a directed graph.

We construct a directed graph where each vertex represents a module and each edge an instantiation. We draw an edge from module A to module B iff module A instantiates module B. If module A instantiates module B twice or more, then
two or more edges going from A to B are drawn. Each module is labelled with the names of its parameters while edges are labeled with: 1) The expressions passed as parameter values during the instantiation and 2) The instantiation guard (if any). The instantiation guard is the conjunction of all the constraints known to be true at the point of the instantiation. This includes parameter constraints and conditions implied from the surrounding generative conditionals.

Figure 5.2 shows the module instantiation graph for the ripple pattern described above. The description includes a single module ripple that contains only two instantiations. First, it has an instantiation of module circ, and second, it has a recursive instantiation of module ripple. This is why the graph shows exactly two edges both originating at the ripple module vertex. One of the edges connects it to the circ vertex and the other is a loop back. The ripple vertex is labeled with $N$ and the other vertex is not labelled since circ is not a parameterized module. The edge going to circ is labelled with $N \geq 1$ which is the only constraint known to be true at its instantiation (due to the parameter constraint on $N$). Again, since circ is not a parameterized module, there are no parameter values on that edge. The other edge however is labeled with $N - 1$ as the parameter value passed at instantiation time and with $N \geq 1 \land N \neq 1$ as the instantiation guard. The left hand side of the conjunction comes from the parameter constraint and the right hand side is known because the instantiation is in the alternative block of the conditional expression.
checking for $N = 1$.

![Diagram](image)

**Figure 5.2** Instantiation Graph for the Ripple Pattern

The first step of the algorithm after constructing the graph is to detect all cycles. If no cycles are detected, then there are no recursive definitions at all and termination is trivially guaranteed. For each cycle detected, however, termination needs to be proved. In Figure 5.2, we only have one cycle composed of one edge and to prove termination we need to show that its instantiation guard cannot hold forever, or in other words, we need to show that each cycle causes the parameter values used at instantiation to change in a way that will eventually cause the guard to fail. In this example, the guard is the conjunction $N \geq 1 \land N \neq 1$. Showing that either $N \geq 1$
or \( N \neq 1 \) cannot hold forever is enough. To do so we need to track how the parameter values change over time by propagating them through the cycle. Since vertex \textit{ripple} has an incoming edge labelled with \( N - 1 \), we record that the initial parameter value passed to \textit{ripple} when instantiated recursively is \( N - 1 \). This value replaces the parameter \( N \) everywhere which means that the edge should now be labelled with \( N - 1 - 1 \) (or \( N - 2 \)) as shown in Figure 5.3(a). Figure 5.3(b) shows what happens when the \( N - 2 \) is propagated through the edge, replacing the original \( N \) with \( N - 2 \). This means that after one cycle, the parameter value has changed from \( N - 1 \) to \( N - 2 \). Proving that \( N - 2 \) is less than \( N - 1 \) is enough to prove that \( N \) is decreasing. This in turn is the only thing needed to prove that the guard conjunction will eventually fail and that the recursive instantiations will eventually terminate.

![Diagram](image)

\( \text{Guard: } N - 1 \geq 1 \land N \neq 1 \)

\( N - 1 \)

\( \text{Ripple} \)

\( \text{N-1} \)

\( \text{Guard: } N - 2 \geq 1 \land N - 2 \neq 1 \)

\( N - 2 \)

\( \text{Ripple} \)

\( \text{N-2} \)

(a) Initial Value (b) Value After Once Cycle

\textbf{Figure 5.3}

In general, to prove termination of a cycle, we first check that there is at least one guard on one of the edges in the cycle. If none is found then we know that this cycle does not terminate. If we can find at least one guard and prove that it will eventually...
fail to hold then we have proved termination. Again, in order to do so, we need to propagate the parameter values through the cycle and see how this affects the guard. This approach is general enough to handle mutually recursive module instantiations. We illustrate this by looking at the sample cycle in Figure 5.4.

**Figure 5.4** A Cycle in the Instantiation Graph Indicating Mutually Recursive Modules

In this cycle a single guard \((N > 10)\) exists on module B's instantiation by module A. To prove termination we need to prove that \(N\) decreases in each cycle. The first time module A is instantiated, \(N\) is set to \(P/3\) and \(M\) is set to \(P/2\). Given these values, we know that if module B is instantiated, it will be instantiated with \(P/3\) and subsequently module C will be instantiated with \(2 \times P/3\) and then module A again with \(N\) equals to \(2 \times P/9\). Since \(2 \times P/9\) is less than \(P/3\), then we have proved that the value of \(N\) decreases which means that eventually it will fail to satisfy the guard guaranteeing the termination of this recursive cycle.
5.6 Implementation

VPP accepts descriptions using the Verilog extensions suggested in this chapter, and its type checker carries out all the analyses described. It relies on Yices to prove termination by proving that the expression obtained after propagating the parameters around the cycle is less than (or greater than depending on the guard) the initial expression. VPP, however, does not use any recurrence solvers at the moment. Instead it reports the gate count as a recurrence relation.

5.6.1 Experimental Results

Increasing Verilog's generative power by introducing the above constructs would not be viable if the generated circuits were less efficient than their manually written counterparts. To experimentally verify that this is not the case, we used the ripple pattern above to generate a 16-bit ripple adder. After elaboration we ended up with a purely structural Verilog description of a 16-bit ripple adder built from only 2 components: a full adder and a 15-bit ripple adder. The 15-bit ripple adder was itself defined in terms of a full adder and a 14-bit ripple adder and so on. The generated description seemed much longer and much more complex than the manual implementation of a 16-bit adder composed of 16 full adders. Although described very differently, both 16-bit adders have exactly the same structure and are expected to need exactly the same amount of resources and to have identical delays. To verify this experimentally, we synthesized both adders using Xilinx ISE and we found that
both required 32 four-input look up tables (LUTs) utilizing 18 slices of the target FPGA. Both circuits had 18 levels of logic with a delay of 33.378 nanoseconds. We also obtained almost identical netlist files when we tried synthesizing both adders using the Icarus Verilog compiler.

5.7 Summary

We extended Verilog’s generative constructs to include the most important features from functional hardware description languages, and we believe that the newly available language is fairly expressive. We were successfully able to generalize our static analyses to support the extended language demonstrating that, by having the right kind of abstractions, a language can be both expressive and predictable. As opposed to RTL and algorithmic level abstractions, the proposed abstractions can be statically checked and do not limit the designer’s full control over the circuits he is creating.
Chapter 6
Conclusion

6.1 Recap of Results

Chapter 2 has argued the pressing need for expressive, well-defined preprocessing constructs in hardware description languages, and showed that a hardware description language with such constructs can be understood as a statically typed two-level language. In this chapter, we focused on one of the most basic properties of a circuit description, namely that it corresponds to a synthesizable circuit. We presented Featherweight Verilog (FV), a core calculus (syntax, type system, and preprocessing semantics) that shows how preprocessing constructs can be defined and reasoned about in the context of a mainstream hardware description language (Verilog). We formalized three technical properties that capture the key features of our calculus, and proved that well-typed FV programs can be successfully elaborated into well-typed, obviously synthesizable circuit descriptions.

In Chapter 3, 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 this chapter) allows designers to safely describe circuit families using structural and generative constructs.

In Chapter 4, we presented a method for statically computing gate counts for generic circuit descriptions. The method is proven correct and was tested on a suite of case studies. Parametric gate count expressions can be immediately used by hardware designers enabling them to better understand the effects of modifying any part of the design on the number of gates used.

Finally, in Chapter 5, we proposed to add recursive modules, higher order modules and polymorphism to Verilog's generative constructs. These relatively simple additions augmented with a couple of practical extensions tremendously increase the expressivity of Verilog as it allows it to describe non-linearly structured circuit families. It also allows for more reusability through the possibility to define parametrized connection patterns. We have also shown that the static analyses that we developed for already existing generative constructs can be extended to support the added constructs.

6.2 Lessons Learned

Overall, this research has revealed several important insights about Verilog, hardware description languages, and programming languages in general.
First, we found out that Verilog is a perfect candidate to be treated and formalized as an STTL due to its elaboration phase and generative constructs. The inexpensive, unobtrusive static type checking associated with the STTL formalization is very useful in detecting incorrect usages of the generative constructs as well as in revealing various inconsistencies that otherwise might have been hidden until later phases. The same framework also allows for parametric gate count calculation (and potentially other estimates) which can help designers to rapidly explore the design space creating better circuits more efficiently.

Second, this work revealed that Verilog is also a natural candidate for usage with dependent types. Dependent types offer powerful static checking but are usually not very appealing from a programmer's point of view as they require the addition of annotations. With Verilog, this is not the case since all the needed information such as array sizes and types is already there waiting to be used.

We acknowledge that Verilog might not be the best possible hardware description language due to its verbosity and its lack of more advanced abstraction mechanisms. While Verilog's verbosity is hard to cure without dramatically changing the language, the lack of advanced abstraction mechanisms can be fixed by adding the desired ones in a backward compatible way. The key here is to pick the right abstraction mechanisms that would make up for Verilog's expressivity limitations while at the same time still give the designer enough control over the generated circuit. Once
more, the STTL framework makes such an extension natural since it allows for adding any reasonably selected constructs and expanding them away at the source level. Additionally, since the STTL framework formally defines how this expansion is carried out, it makes sure that the designer has total control and it allows us to add typing rules providing the required static analyses and resulting guarantees.

We strongly believe that behavioral constructs are extremely useful abstraction mechanisms for simulations and verification, but they are not ideal for synthesis. On the other hand structural constructs are always guaranteed to synthesize exactly as expected. That is why we believe that abstraction mechanisms should ideally be built (with clearly defined semantics) on top of structural constructs. This is exactly what is provided through generative constructs. In brief, if designed correctly, generative constructs can provide the necessary levels of abstraction without giving away the designer's control over the generated hardware.

The results obtained here are applicable to any language with generative features (either explicit or implicit). In general, we found the STTL framework to be particularly suitable for formalizing and implementing generative features of languages. It also offers a very convenient way to extend an existing language in a safe and backward compatible way.
6.3 Future Work

There is always room for improving a language further. In the same line of research, there are two primary directions for pushing Verilog forward.

First, we would like to continue exploring new generative constructs. In order to do this we would like to encourage circuit designers to use VPP and give us feedback about what kind of circuits still remain hard to express. Example features that we would like to add include parameter currying and support of algebraic data types.

Second, we would like to extend our static analysis to check for additional circuit properties and provide more guarantees. One such example is the ability to statically compute a parametric circuit delay for a family of circuits. Verifying that the number of connections to a gate's output does not exceed its fan-out or that the circuit is free from short-circuited output pins are some examples of useful guarantees that one would like to have.
References


52. Opencores.org, “Or1200’s 32x32 multiply for asic.” http://www.opencores.org/cvsweb.shtml/or1k/or1200/rtl/verilog/or1200_amulp2.32x32.v.


Appendix A
Case Studies

This appendix contains the listings of some VPP examples used throughout the thesis. These examples demonstrate how to write purely structural descriptions of circuit families with the help of generative constructs.

A.1 Bus Inverter

```verbatim
1 module invtbn (x, y);
2    parameter N=4;
3    input [N-1:0] y;
4    output [N-1:0] x;
5    genvar i;
6
7 for (i=0; i < N; i=i+1)
8    not(x[i], y[i]);
9 endmodule
```

A.2 Linear Parity Checker

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

A.3 Ripple Adder

module full_adder (sum, cout, a, b, cin);
  input a, b, cin;
  output sum, cout;
  wire w1, w2, w3;

  xor(w1, a, b);
  xor(sum, w1, cin);
  and(w2, cin, w1);
  and(w3, a, b);
  or(cout, w2, w3);
endmodule

module adder (s, cout, a, b, cin);
  parameter N=8;
  input [N-1:0] a, b;
  input cin;
  output [N-1:0] s;
  output cout;
  wire [N:0] c;
  genvar i;

  assign c[0] = cin;
  for (i=0; i < N; i=i+1)
    full_adder fa (s[i], c[i+1], a[i], b[i], c[i]);
  assign cout = c[N];
endmodule

A.4 Carry Select Adder Block

module full_adder (sum, cout, a, b, cin);
  input a, b, cin;
  output sum, cout;
  wire w1, w2, w3;

  xor(w1, a, b);
  xor(sum, w1, cin);
  and(w2, cin, w1);
  and(w3, a, b);
  or(cout, w2, w3);
endmodule

module ripple_adder(s,cout,a,b,cin);
  parameter N=8;
  input [N−1:0] a,b;
  input cin;
  output [N−1:0] s;
  output cout;
  wire [N:0] c;
  genvar i;

  assign c[0] = cin;
  for (i=0;i < N;i=i+1)
    full_adder fa (s[i],c[i+1],a[i],b[i],c[i]);
  assign cout=c[N];
endmodule

module mux(mout,a,b,sel);
  output mout;
  input a,b,sel;
  wire w1,w2;
  or (mout,w1,w2);
  and (w1,a,"sel);
  and (w2,b,sel);
endmodule

module carry_select_adder_block(sum,cout,a,b,cin);
  parameter N=4;
  input [N−1:0] a,b;
  input cin;
  output [N−1:0] sum;
  output cout;
  wire [N−1:0] sum0,sum1;
  wire cout0,cout1;
  genvar i;

  ripple_adder #(N) r1 (sum0,cout0,a,b,0);
  ripple_adder #(N) r2 (sum1,cout1,a,b,1);
for(i=0;i < N;i=i+1)
  mux sum_mx (sum[i],sum0[i],sum1[i],cin);
  mux cout_mx (cout,cout0,cout1,cin);
endmodule

A.5 Counter

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

A.6 Decoder

module decoder (dec_out,dec_in);
  parameter  N=3;
  input [N-1:0]   dec_in;
  output [2*N-1:0] dec_out;
  wire [N-1:0]    ndec_in;
  wire [N-1:0]    temp [2*N-1:0];
  genvar    i,j;

for(i=0;i < N;i=i+1)
  not (ndec_in[i],dec_in[i]);
for (i=0; i < 2**N; i=i+1) begin
    for (j=0; j < N; j=j+1) begin
        if ((i >> j) % 2==0)
            assign temp[i][j]=ndec_in[j];
        else
            assign temp[i][j]=dec_in[j];
    end
    assign dec_out[i] = &temp[i];
end
endmodule

A.7 Multiplexer

assume decoder #(N 2)
(output wire [2**N] a,
input wire [N] b) 2**N+N;

module gen_mux(mux_out, mux_in, sel);
    parameter M=3;
    input [2**M-1:0] mux_in;
    input [M-1:0] sel;
    output mux_out;
    genvar i;
    wire [2**M-1:0] decsel;
    wire [2**M-1:0] p;
    decoder #(M) dec1 (decsel, sel);
    generate
        for (i=0; i<2**M; i=i+1)
            and (p[i], decsel[i], mux_in[i]);
    endgenerate
    assign mux_out = |p;
endmodule

A.8 Multiplier

assume adder #(N 4)
(output wire [N] sum,
output wire count,
input wire [N] a,
input wire [N] b,
input wire cin) 5*N;

module multiplier(c, a, b);

parameter N=6;
parameter M=3;
input [N-1:0] a;
input [M-1:0] b;
output [N+M-1:0] c;
wire [N-1:0] tmpand [M-1:0];
wire [N-1:0] tmpsum [M-1:0];
wire tmpc [M-2:0];
genvar i, j;

for (i=0; i < N; i=i+1)
  for (j=0; j < M; j=j+1)
    and (tmpand[j][i], a[i], b[j]);

assign c[0] = tmpand[0][0];
assign tmpsum[0] = tmpand[0];
assign tmpc[0] = 0;
assign c[0] = tmpsum[0][0];

for (i=0; i < M-1; i=i+1) begin
  adder #(N) add1 (tmppsum[i+1], tmpc[i+1], tmpand[i],
  {tmpc[i], tmppsum[i][N-1:1]}, 0);
  assign c[i+1] = tmppsum[i+1][0];
end
assign c[N+M-1:M] = {tmpc[M-1], tmppsum[M-1][N-1:1]};
endmodule
Appendix B
VPP User Manual

This appendix briefly explains how to install and use VPP. VPP has been tested on Linux and Mac OS X (versions 10.4 and 10.5). Although we have not tried it under Windows, we believe it should compile without problems under Cygwin.

B.1 Prerequisites

In order to use VPP, you need to compile it first. Currently we do not distribute precompiled binaries of VPP. To successfully compile VPP, you need to download and install the following:

- OCaml (available from http://caml.inria.fr/)

B.2 Installation

After installing VPP’s prerequisites, you need to download it and build it by following these steps:


2. Extract the downloaded tarball file

3. Edit the first two lines of the existing Makefile to reflect the correct paths of Yices and OCaml based on your system’s configuration
4. Issue a `make world` command

B.3 Invocation and Command-line Arguments

Once VPP was successfully compiled, you should be able to run it from the command prompt by issuing a `./vpp options <filename>` command. Typically you will want to use it by typing something similar to `./vpp -ppg -o test_out.v test_in.v`. This will invoke VPP's type checker on the input file `test_in.v`. If no type errors are found, VPP will elaborate the description and save the result in the output file `test_out.v`. Also because of the `-ppg` flag, VPP will print the parametric gate count of each module to the standard output.

If you type `./vpp -help`, you will get:

```
vpp options <filename>
Verilog Preprocessor ver 1.0
-tc type check only
-ppg print parametric gate counts
-time print timings
-o output filename for expanded Verilog (only works when -tc is off)
-help Display this list of options
--help Display this list of options
```
Appendix C
Preprocessing Errors

Defining abnormal cases is equally important. Figures C.1 and C.2 show when errors can occur during preprocessing. Namely, when any of the following situations occur:

1. Elaborating a circuit description:
   
   (a) The main module is not defined (E-PE1).
   
   (b) An error occurs while elaborating its body given all other module definitions (E-PE2).

2. Elaborating the main module body:
   
   (a) The main module has a non-empty parameter sequence (E-BE1).
   
   (b) Elaborating any of the parallel statements in the body generates an error (E-BE2).

3. Elaborating a module instantiation:
   
   (a) Any expression passed as a parameter fails to evaluate (E-ME1).
   
   (b) Any LHS expression passed as a port connection fails to elaborate (E-ME2).
(c) There is no corresponding module definition (E-ME3).

(d) The number of parameters passed is not correct (E-ME4).

(e) The number signals passed is not correct (E-ME5).

(f) An error occurs while elaborating any of the parallel modules composing the body of the instantiated module (E-ME6).

4. Elaborating an assignment:

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

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

5. Elaborating a conditional statement:

   (a) The conditional expression cannot be evaluated (E-IfE).

   (b) The conditional expression evaluates to true and any of the parallel statements of the consequent cannot be elaborated (E-IfTrueE).

   (c) The conditional expression evaluates to false and any of the parallel statements of the alternative cannot be elaborated (E-IfFalseE).

6. Elaborating loops:

   (a) The initialization expression fails to evaluate (E-ForE1).

   (b) The condition expression fails to evaluate (E-ForE2).
(c) Any parallel statement in the body of the loop fails to elaborate (E-ForE3).

(d) The remaining loop iterations fail to elaborate (E-ForE4).

7. Elaborating an expression:

(a) It is a signal indexed by one or more expressions that fail to evaluate (E-IdxE1E and E-Rg1E).

(b) It is a parameter name (E-Par1E).

(c) It is composed of operations on expressions where at least one fails to elaborate (E-Op1E).

8. Evaluating an expression:

(a) It is a signal (E-IdE, E-IdxE0E and E-Rg0E).

(b) It is a parameter name (E-Par0E).

(c) It is composed of operations on expressions including at least one that fails to evaluate (E-Op0E).

It is important to note that, because we use substitution to eliminate level 0 variables, encountering any identifier during evaluation constitutes a preprocessing error. This formalizes the property that any dependency on either an uninstantiated parameter or a wire value constitutes a preprocessing error.
Figure C.1  Operational Semantics Errors For Parallel Statements
\[
\begin{align*}
[1] &\quad \frac{p \downarrow p_\bot}{(D_i) m \downarrow \text{err}} \quad \text{module } m \, b \notin \langle D_i \rangle \quad \text{(E-PE1)}
\quad \frac{(D_i) + b \downarrow \text{err}, \emptyset}{(D_i) m \downarrow \text{err}} \quad \text{module } m \, b \in \langle D_i \rangle \quad \text{(E-PE2)}
\end{align*}
\]

\[
\begin{align*}
(D) \vdash b \downarrow \bot, b_\bot, (D) &\quad \text{if } I \neq \emptyset \\
(L) &\quad \frac{(D_i) \vdash \langle x_i \rangle \in I \langle d_j \, s_j \rangle \, \text{is} \, \langle b_q \, b_\bot \rangle \langle f_k \rangle \downarrow \text{err}, \emptyset}{(D_i) \vdash \emptyset \langle d_j \, s_j \rangle \, \text{is} \, \langle b_q \, b_\bot \rangle \langle f_k \rangle \downarrow \text{err}, \emptyset} \quad \text{(E-BE1)}
\quad \text{(E-BE2)}
\end{align*}
\]

\[
\begin{align*}
I &\downarrow L_\bot \\
\text{See } e &\downarrow e_\bot
\end{align*}
\]

\[
\begin{align*}
e &\downarrow e_\bot \\
\frac{e_0 \downarrow \text{err}}{s[e] \downarrow \text{err}} \quad \text{(E-Idx1E)}
\quad \frac{\exists i. e_i \downarrow \text{err}}{s[e_1 : e_2] \downarrow \text{err}} \quad \text{(E-Rg1E)}
\quad \frac{x \downarrow \text{err}}{x \downarrow \text{err}} \quad \text{(E-Par1E)}
\end{align*}
\]

\[
\begin{align*}
e &\downarrow e_\bot \\
\frac{s_0 \downarrow \text{err}}{s[e] \downarrow \text{err}} \quad \text{(E-IdE)}
\quad \frac{s[e]_0 \downarrow \text{err}}{s[e_1 : e_2] \downarrow \text{err}} \quad \text{(E-Idx0E)}
\quad \frac{x_0 \downarrow \text{err}}{x_0 \downarrow \text{err}} \quad \text{(E-Rg0E)}
\end{align*}
\]

\[
\begin{align*}
\frac{\exists i. e_i \downarrow \text{err}}{f(e_i) \downarrow \text{err}} \quad \text{(E-OpE)}
\quad \frac{x_0 \downarrow \text{err}}{f(e_i) \downarrow \text{err}} \quad \text{(E-OpE)}
\end{align*}
\]

**Figure C.2** Operational Semantics Errors For All Other Constructs
Appendix D
Proofs and Auxiliary Results

D.1 For Chapter 2

In this section we provide the complete proof for theorems appearing in Chapter 2.

D.1.1 Substitution Lemma

To prove the type preservation theorem, we first need a substitution lemma. We only need to define substitution on parallel statements. Therefore we state the substitution lemma as follows:

**Lemma 3** (Substitution). If \( \Delta; \Gamma, x : d t \vdash P \) and \( \Gamma \vdash^n v : d t \) then \( \Delta; \Gamma \vdash P[x \mapsto v] \)

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

We also need to establish the same property for smaller constructs. We omit the proofs of these auxiliary lemmas since they are identical to the previous one. We only need to state Lemmas 4, and 5 that establish that substitution preserves typing for \( e \) and \( l \) respectively.
**Lemma 4** (Substitution for expressions). If $\Gamma, x : d_1 \ t_1 \vdash_{\eta_2} e : d_2 \ t_2$ and $\Gamma \vdash_{\eta_1} v : d_1 \ t_1$ then $\Gamma \vdash_{\eta_2} e[x \mapsto v] : d_2 \ t_2$

**Lemma 5** (Substitution for LHS values). If $\Gamma, x : d_1 \ t_1 \vdash_{\eta_2} l : d_2 \ t_2$ and $\Gamma \vdash_{\eta_1} v : d_1 \ t_1$ then $\Gamma \vdash_{\eta_2} l[x \mapsto v] : d_2 \ t_2$

D.1.2 Type Preservation

We now proceed to proving the type preservation theorem.

Lemmas 6, 7, 8, 9, and 10 establish that preprocessing preserves typability for $e$ (2 lemmas), $l$, $P$, and $b$ terms. They are all required to prove Theorem 1

**Lemma 6** (Type preservation for level 0 expressions). If $\Gamma^+ \vdash^0 e : d \ t$ and $e \xrightarrow{\eta} e'$ then $\Gamma^+ \vdash^0 e' : d \ t$

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

**Lemma 7** (Type preservation for level 1 expressions). If $\Gamma^+ \vdash^1 e : d \ t$ and $e \xrightarrow{\eta} e'$ then $\Gamma^+ \vdash^1 e' : d \ t$
Proof. We proceed by induction on the evaluation tree of $e$. We have six cases:

- **Case Id**: $e$ is $s$ and from E-Id1 $e'$ is $s$. Therefore the lemma holds trivially.

- **Case Index**: $e$ is $s[e_1]$ and from E-Index1 $e'$ is $s[v]$ where $e_1 \xrightarrow{0} v$. From T-Index we know that $\Gamma^+ \vdash^0 e_1 : \{\text{in}\} \text{ int}$. From lemma 6, we get that $\Gamma^+ \vdash^0 v : \{\text{in}\} \text{ int}$. Using T-index again we get $\Gamma^+ \vdash^1 s[v] : d\ t$.

- **Case Range**: $e$ is $s[e_1 : e_2]$ and from E-Range1 $e'$ is $s[v_1 : v_2]$. Following the same steps of the previous case but using T-Range instead of T-Index and using lemma 6 twice we get the desired result.

- **Case Param**: $e$ is $x$. The lemma hold vacuously because from T-Param it is clear that $e$ cannot be typed using $\Gamma^+$.

- **Case Int**: $e$ is $v$ and from E-Int1 $e'$ is $v$. Therefore the lemma holds trivially.

- **Case Op**: $e$ is $f(e_i)$ and from E-Op1 $e'$ is $f(e'_i)$. From T-Op we know that $\{\Gamma^+ \vdash^1 e_i : d' t_i\}$ and using the induction hypothesis on $e_i$ we get $\{\Gamma^+ \vdash^1 e'_i : d' t_i\}$. Using T-Op again, we obtain the required result.

\[\square\]

**Lemma 8** (Type preservation for LHS values). If $\Gamma^+ \vdash^1 l : d\ t$ and $l \xrightarrow{1} l'$ then $\Gamma^+ \vdash^1 l' : d\ t$

Proof. This follows directly from Lemma 7 because any $l$ is also an $e$. \[\square\]
Lemma 9 (Type preservation for parallel statements). If $\Delta; \Gamma^+ \vdash P$ and $\langle D_i \rangle \vdash P \overset{1}{\rightarrow} \langle P'_j \rangle, \langle D_k \rangle$ then $\vdash \langle D_k \rangle : \Delta'$ and $\{\Delta'; \Gamma^+ \vdash P'_j\}$

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

- **Case Module Instantiation:** From E-Mod, we know that preprocessing $P$ gives a sequence of parallel statements composed of one parallel statement $P'$ equal to $m'\langle l_j \rangle$ and a set of modules $\langle \text{module } m' \langle \rangle \}\langle d_j s_j \rangle$ $\bigcup_{r \in R(k)} \langle P_r \rangle$ $\bigcup_{h \in H(k)} \langle D_h \rangle$. The preprocessing of each $P_k\{[x_i \mapsto v_i]\}$ is a subtree in the preprocessing of $P$ and since we know from T-Mod that $\Delta(m) = |\langle e_i \rangle |\langle d_j \rangle$, we also know that $m$ is typed which implies from T-Body that every $P$ in $\langle P_k \rangle$ is also typed. Therefore we know from Lemma 3 that $P_k\{[x_i \mapsto v_i]\}$ is also typed. From the induction hypothesis we know that all parallel statements $\bigcup_{r \in R(k)} \langle P_r \rangle$ and module definitions $\bigcup_{h \in H(k)} \langle D_h \rangle$ 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 $\Delta'$ contains the type of $m'$ in addition to the types of all the generated modules. From T-Mod, we can easily get that $m'\langle l_j \rangle$ is well-typed under $\Delta'$. 
• **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 assign $l' e'$ and an empty set of modules. From T-Assign we know that both $l$ and $e$ are well-typed under $\Gamma^+$ and using Lemmas 8 and 7 we can conclude that $l'$ and $e'$ have the same types under $\Gamma^+$ therefore using T-Assign again we reach $\Delta; \Gamma^+ \vdash \text{assign } l' e'$. 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 $\bigcup_k (P_k)_{r \in R(k)}$, and a sequence of modules $\bigcup_k (D_k)_{h \in H(k)}$. From T-If, we know that $\{\Delta; \Gamma^+ \vdash P_k\}$ and since each parallel statement in $P_k$ is smaller than $P$, we can apply the induction hypothesis and conclude that all the parallel statements and the modules generated from the preprocessing of each of them are well-typed.

  - **Case False:** Similarly we can show that all parallel statements and modules generated from the preprocessing of each parallel statement in $(P_j)$ 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 $\bigcup_k (P_r)^{re,R(k)} \cup (P_j)$, and a sequence of modules $\bigcup_k (D_h)^{he,H(k)} \cup (D_q)$ 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 6 that $\Gamma^+ \vdash^0 v_1 : \{\text{in}\} \text{ int}$. We also know that $\{\Delta; \Gamma^+, y : \{\text{in}\} \text{ int} \vdash P_k\}$ and that $\Gamma^+, y : \{\text{in}\} \text{ int} \vdash^0 e_2, e_3 : \{\text{in}\} \text{ int}$. Using Lemmas 3 and 4 respectively we get $\{\Delta; \Gamma^+ \vdash P_k[y \mapsto v1]\}$ and that $\Gamma^+ \vdash^0 e_2[y \mapsto v1], e_3[y \mapsto v1] : \{\text{in}\} \text{ int}$. Now using Lemma 6 we know that $\Gamma^+ \vdash^0 v_2 : \{\text{in}\} \text{ int}$. and using the induction hypothesis we get $\vdash \bigcup_k (D_h)^{he,H(k)} : \Delta_1$ and $\forall k. \Delta; \Gamma^+ \vdash (P_r)^{re,R(k)}$. Finally using T-For again we get that $\{\Delta; \Gamma^+ \vdash \text{for}(y = e_3[y \mapsto v1]; e_2; y = e_3) P_k\}$ which by the induction hypothesis shows that $\vdash (D_q) : \Delta_2$ 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.

\[\Box\]

**Lemma 10** (Type preservation for main module body). If $\Delta \vdash b : M$ and $(D_k) \vdash b \triangleright^1 b', (D_k)$ then $(D_k) : \Delta'$ 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 \( \langle d_j \cdot s_j \rangle \) is \( \langle t_q \cdot s_q \rangle \) \( \bigcup \langle P_r \rangle_{r \in R(k)} \) and a set of modules \( \bigcup \langle D_h \rangle_{h \in H(k)} \). We also know from T-Body that each parallel statement \( P \) in \( P_k \) is typed under \( \Delta \) and \( \langle s_j : d_j \cdot \text{wire} \rangle \uplus \langle s_k : \{\text{in, out}\} \cdot t_k \rangle \), therefore using Lemma 9 we can conclude that \( \bigcup \langle P_r \rangle_{r \in R(k)} \) and \( \bigcup \langle D_h \rangle_{h \in H(k)} \) 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 \( \langle D_r \rangle \uplus \langle \text{module } m \ b' \rangle m \). We also know that \( \langle D_i \rangle \vdash b \leftrightarrow b', \langle D_r \rangle \) 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 \( \Delta \) 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 10 we can conclude that \( \vdash \langle D_r \rangle : \Delta' \) and \( \Delta' \vdash b' : M \). This also means that \( \langle D_r \rangle \uplus \langle \text{module } m \ b' \rangle \) is well-typed under the empty environment (T-MSeq). This proves that \( \vdash p' \).
D.1.3 Type Safety

Proving the Type Safety Theorem (Theorem 2) is trivial.

Proof. This result follows directly from Theorem 1 since no typing rules will consider \texttt{err} well-typed.

D.1.4 Preprocessing Soundness

Lemmas 11, 12, 13, 14, and 15 establish that the result of preprocessing \( e \) (2 lemmas), \( l, P \), and \( b \) terms respectively produces a fully preprocessed term of the same category. Theorem 3 establishes this property for circuit descriptions.

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

Lemma 11 (Preprocessing soundness for level 0 expressions). If \( e \stackrel{0}{\rightarrow} e' \) then \( e' \in \hat{e} \).

Proof. We proceed by induction on the evaluation tree of \( e \). E-Int0 case is immediate since \( e' = v \) which is allowed in \( \hat{e} \). Case E-Op0 is also true because from the induction hypothesis each \( u_i \) is in \( \hat{e} \) and therefore the result of applying an operator \( f \) on \( \langle u_i \rangle \) will be in \( \hat{e} \).

Lemma 12 (Preprocessing soundness for level 1 expressions). If \( e \stackrel{1}{\rightarrow} e' \) then \( e' \in \hat{e} \).

Proof. We proceed by induction on the evaluation tree of \( e \). E-Id1 case is immediate since \( e' = s \) which is allowed in \( \hat{e} \). Cases E-Index1, E-Range1 make use of Lemma 11.
to make sure that $e' = s[u]$ and $e' = s[v_1 : v_2]$ are in $\hat{e}$. Case E-Int1 is immediate.

Finally, case E-Op1 makes use of the induction hypothesis to prove its goal. □

Lemma 13 (Preprocessing soundness for LHS values). If $l \rightarrow l'$ then $l' \in \hat{l}$

Proof. Follows directly from Lemma 12 □

Lemma 14 (Preprocessing soundness for parallel statements). If $\langle D_i \rangle \vdash P \rightarrow \langle P'_i \rangle, \langle D_k \rangle$ then $\{P'_j \in \hat{P}\}$ and $\{D_k \in \hat{D}\}$

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

- **Case Module Instantiation**: From E-Mod, we know that preprocessing $P$ gives a sequence of parallel statements composed of one parallel statement $P'\text{equal to } m'(\langle t_j \rangle)$ and a set of modules $\langle \text{module } m' \rangle \langle d_j s_j \rangle$ is $\langle t_q s_q \rangle$

$$\bigcup_{k} \langle P_r \rangle \cup_{k} \langle D_h \rangle$$

$P'$ is clearly in $\hat{P}$ while the generated set of modules is in $\hat{D}$ if all the parallel statements composing their bodies are in $\hat{P}$. The preprocessing of each $P_k[[x_1 \mapsto v_i]]$ is a subtree in the preprocessing of $P$ so we know from the induction hypothesis that each of them results in a sequence of parallel statements in $\hat{P}$. Similarly, any modules produced in the process are in $\hat{D}$ by the inductive hypothesis. Note that we know that every $v_i$ is in $\hat{e}$ from Lemma 11.

- **Case Assign**: From E-Assign, we know that preprocessing $P$ gives a sequence of parallel statements composed of one parallel statement $P'$ equal to
assign \(l' \ e'\) and an empty set of modules. Using Lemmas 13 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 preprocessing result is equal to a sequence of parallel statements and a sequence of modules. These sequences are generated from preprocessing each of the parallel statements in \(\langle P_k \rangle\). Since these are parallel statements having smaller evaluation trees than \(P\), we can use the induction hypothesis to show that the results are sound.
  
  - **Case False**: Similarly it can be seen from E-IfFalse, that preprocessing each parallel statement in \(\langle P_j \rangle\) produces fully preprocessed terms.

- **Case For**: We have two subcases to consider here:
  
  - **Case False**: From E-ForFalse, The preprocessing result is equal to an empty sequence of parallel statements and an empty sequence of modules.
  
  - **Case True**: From E-ForTrue, The preprocessing result is equal to a sequence of parallel statements and a sequence of modules. This sequence is generated from preprocessing the body of the for loop after substitution and preprocessing another for statement. Since both the body of the loop and the new loop are parallel statements having smaller evaluation trees
than $P$, we can use the induction hypothesis to show that the results are sound. Again, note that substitutions in E-ForTrue do not introduce any subterms that are not in our preprocessed syntax since we are substituting using $v_1$ which is obtained by evaluating $e_1$ at level 0 and is therefore in $\hat{e}$ as proved in Lemma 11.

\[\Box\]

**Lemma 15** (Preprocessing soundness for main module body). If $(D_r) \vdash b \overset{1}{\rightarrow} b', (D_k)$ then $b' \in \hat{b}$ and $\{D_k \in \hat{D}\}$

**Proof.** From E-Body, we can see that the result from preprocessing the body $b$ of the main module gives a new body $b'$ equal to $(\langle d_j, s_j \rangle \mid t_q, s_q \rangle \cup \bigcup_{r \in R(k)} \langle P_r \rangle)^{k \in H(k)}$ and a set of modules $\bigcup_{k \in H(k)} \langle D_h \rangle$. Note that $b'$ is parameter free and that from Lemma 14 we know that all parallel statements used to construct $\bigcup_{k \in H(k)} \langle P_r \rangle$ are in $\hat{P}$ and that all module definitions used to construct $\bigcup_{k \in H(k)} \langle D_h \rangle$ are in $\hat{D}$ because they all come from parallel statements preprocessing. Therefore $b' \in \hat{b}$ and $\{D_k \in \hat{D}\}$ are true. \[\Box\]

Finally, we prove the Preprocessing Soundness Theorem (Theorem 3) as follows:

**Proof.** From E-Prog, $p'$ is $(D_r) \cup \langle \text{module } m \ b' \rangle m$ where $(D_r) \vdash b \overset{1}{\rightarrow} b', (D_r)$ and $b$ is the body of the main module of $p$. From Lemma 15 we know that for every $D_r$, we have $D_r \in \hat{D}$ and that $b' \in \hat{b}$, therefore $p'$ is in $\hat{p}$. \[\Box\]
D.2 For Chapter 3

In this section we provide the complete proof for theorems appearing in Chapter 3

D.2.1 Substitution Lemma

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 16 (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 $\Delta; \Gamma[x \mapsto v] ; C \vdash P[x \mapsto v]$

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 17 (Substitution for constraints). If $C \triangleright c$ and $C, x = v$ is satisfiable then $C \triangleright c[x \mapsto v]$

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 $C \triangleright c[x \mapsto v]$ still holds. $\square$

To prove Lemma 16, we need to establish the same property for smaller constructs first. We first state and prove Lemmas 18, and 19 that establish that substitution preserves typing for $e$ and $l$ respectively.
Lemma 18 (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 $(\Gamma, x : d_1 t_1)(s) = d_2 t_2(e_i, e_u)$, that $\Gamma, x : d_1 t_1; C \vdash^0 e_1 : \{\text{in}\} \text{ int}$, that $C \triangleright e_1 \geq e_i$ and that $C \triangleright e_1 \leq e_u$. Since $e_1$ has a smaller typing derivation tree than $e$, we can apply the induction hypothesis to get that $\Gamma[x \mapsto v]; C \vdash^0 e_1[x \mapsto v] : \{\text{in}\} \text{ int}$ and again since $s$ cannot be equal to $x$ than $\Gamma(s) = d_2 t_2(e_i, e_u)$. This also means that $(\Gamma[x \mapsto v])(s) = d_2 t_2[x \mapsto v](e_i[x \mapsto v], e_u[x \mapsto v])$ Applying Lemma 17 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 $s[e_1[x \mapsto v]]$. 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:

  - $e$ is $x$: In this case $e[x \mapsto v]$ is $v$ from the substitution rule. From T-Par we get that $d_2 t_2 = \{\text{in}\} \text{ int}$. But from T-Int we know that $\Gamma; C \vdash^{n_2} v : \{\text{in}\} \text{ int}$. And we know that $\text{int}[x \mapsto v] = \text{int}$. 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 $(\Gamma, x : d_1 t_1)(y) = \{\text{in}\} \text{ int}$. But since $y \neq x$ then $\Gamma(y) = \{\text{in}\} \text{ int}$. 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 $\Gamma$.

- **Case Op**: $e$ is $f(e_i)$ and $e[x \mapsto v]$ is $f(e_i[x \mapsto v])$ from the substitution rule. From T-Op, we know that $d_2 t_2 = \{\text{in}\} \Sigma_i (e_i) ((f, n_2, (t_i)))$ and that $\{\Gamma, x : d_1 t_1; C \vdash^{n_2} e_i : \{\text{in}\} t_i\}$. Since each $e_i$ has a smaller typing derivation than $e$, we can use the induction hypothesis to get $\{\Gamma[x \mapsto v]; C \vdash^{n_2} e_i[x \mapsto v] : \{\text{in}\} t_i[x \mapsto v]\}$. Therefore by using T-Op again, we get that $\Gamma[x \mapsto v]; C \vdash^{n_2} f(e_i[x \mapsto v]) : \{\text{in}\} \Sigma_i (e_i[x \mapsto v]) ((f, n_2, (t_i[x \mapsto v])))$. Noting that $|\langle e_i[x \mapsto v]\rangle| = |\langle e_i\rangle|$ we reach the desired conclusion.
Lemma 19 (Substitution for LHS values). If $\Gamma \vdash_{n_2} l : d_2 \; t_2$ and $\Gamma \vdash_{n_1} v : d_1 \; t_1$ and $C, x = v$ is satisfiable then $\Gamma[x \mapsto v] \vdash_{n_2} l[x \mapsto v] : d_2 \; t_2[x \mapsto v]$

Proof. This follows directly from Lemma 18 because any $l$ is also an $e$. □

Now we have all the auxiliary results needed to prove the Substitution Lemma (Lemma 16). 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 $\{\Gamma, x : d \; t; C \vdash^0 e_i : \{\text{in} \} \; \text{int}\}$, $\Delta(m) = | \langle e_i \rangle | \langle d_j \rangle$, $\{\Gamma, x : d \; t; C \vdash^1 l_j : d_j' t_j'\}$, $\{d_j \subseteq d_j'\}$, and that each $t'_j$ have equal width to the corresponding $t_j\{[x_i \mapsto e_i]\}$ given $C$. From Lemmas 18 and 19 we get that $\{\Gamma[x \mapsto v]; C \vdash^0 e_i[x \mapsto v] : \{\text{in} \} \; \text{int}\}$ and $\{\Gamma[x \mapsto v]; C \vdash^1 l_j[x \mapsto v] : d_j' t_j'[x \mapsto v]\}$ therefore using Lemma 17 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 assign $l \; e$ and from the substitution rule $P[x \mapsto v]$ is 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 19 we get that
\[ \Gamma[x \mapsto v]; C \vdash^1 l[x \mapsto v] : d_1 t_1[x \mapsto v] \] Therefore using Lemma 17 with T-
Assign1 again, we know that \( \Delta; \Gamma[x \mapsto v]; C \vdash \text{assign} \ l[x \mapsto v] \ e[x \mapsto v] \).

\( - \ e \) 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 19 and 18 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 17 with T-
Assign2 again, we know that \( \Delta; \Gamma[x \mapsto v]; C \vdash \text{assign} \ l[x \mapsto v] \ e[x \mapsto v] \).

- **Case If:** \( P \) is if \( e \) then \( \langle P_1 \rangle \) else \( \langle P_2 \rangle \) and from the substitution rule
\( P[x \mapsto v] \) is if \( e[x \mapsto v] \) then \( \langle P_1[x \mapsto v] \rangle \) else \( \langle P_2[x \mapsto v] \rangle \). From T-
If, we know that \( \Gamma, x : d \ t; C \vdash^0 \{ \text{in} \} \text{ int}, \) that \( \{ \Delta; \Gamma, x : d \ t; C, e \vdash P_1 \}, \) and that \( \{ \Delta; \Gamma, x : d \ t; C, \neg e \vdash P_2 \}. \) From Lemma 18 we get that \( \Gamma[x \mapsto v]; C \vdash^0 e[x \mapsto v] : \{ \text{in} \} \text{ int} \) and from the induction hypothesis we get that
\( \{ \Delta; \Gamma[x \mapsto v]; C, e \vdash P_1[x \mapsto v] \} \) and \( \{ \Delta; \Gamma[x \mapsto v]; C, \neg e \vdash P_2[x \mapsto v] \} \) Therefore using T-If again we conclude that \( \Gamma[x \mapsto v]; C \vdash \text{if} \ e[x \mapsto v] \text{ then} \langle P_1[x \mapsto v] \rangle \text{ else} \langle P_2[x \mapsto v] \rangle \)

- **Case For:** \( P \) is for(\( y = e_1; y < e_2; y = y + e_3 \) \( P_1 \)) and from the substitution
rule \( P[x \mapsto v] \) is for(\( y = e_1[x \mapsto v]; y < e_2[x \mapsto v]; y = y + e_3[x \mapsto v] \) \( P_1[x \mapsto v] \)). From T-For, we know that \( \{ \Delta; \Gamma, x : d \ t, y : \{ \text{in} \} \text{ int}; C, y \geq e_1, y < e_2 \vdash P_1 \}, \) that \( \Gamma, x : d \ t, y : \{ \text{in} \} \text{ int}; C \vdash^0 e_3 : \{ \text{in} \} \text{ int}, \) that \( \Gamma, x : d \ t; C \vdash^0 e_1, e_2 :
\{\text{in} \} \text{ int, and that } e_3 \text{ is positive given } C. \text{ From Lemma 18 we get that } \Gamma[x \mapsto v], y : \{\text{in} \} \text{ int}; C \vdash^0 e_3[x \mapsto v] : \{\text{in} \} \text{ int, and } \Gamma; C \vdash^0 e_1[x \mapsto v], e_2[x \mapsto v] : \{\text{in} \} \text{ int and from the induction hypothesis we get that } \{\Delta; \Gamma[x \mapsto v], y : \{\text{in} \} \text{ int}; C, y \geq e_1, y < e_2 \vdash P_1[x \mapsto v]\} \text{ Therefore using T-For again (with Lemma 17) we know that } \Gamma[x \mapsto y]; C \vdash \text{for}(y = e_1[x \mapsto v]; y < e_2[x \mapsto v]; y = y + e_3[x \mapsto v]) (P_1[x \mapsto v]).

\square

D.2.2 Type Preservation

Lemmas 20, 21, 22, 23, and 24 establish that preprocessing preserves typability for \( e \) (2 lemmas), \( l \), \( P \), and \( b \) terms. They are all required to prove Theorem 4.

**Lemma 20** (Type preservation for level 0 expressions). \( \text{If } \Gamma^+; C \vdash^0 e : d \; t \text{ and } e \vdash^0 e' \text{ then } \Gamma^+; C \vdash^0 e' : d \; t \)

*Proof.* We proceed by induction on the evaluation tree of \( e \). We have six cases: Cases Id, Index, Range are vacuously true because none of them can be typed at level 0. T-Par is also vacuously true because \( \Gamma^+ \) 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 \( \Sigma \) returns by definition the same type that is returned when applying the operator \( f \) at level 0. \( \square \)
Lemma 21 (Type preservation for level 1 expressions). If $\Gamma^+; C \vdash^1 e : d \ t$ and $e \xrightarrow{1} e'$ 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 $e_1 \xrightarrow{0} v$. From T-Idx we know that $\Gamma^+; C \vdash^0 e_1 : \{\text{in}\} \text{ int}$. From Lemma 20, we get that $\Gamma^+; C \vdash^0 v : \{\text{in}\} \text{ int}$. 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 20 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 $\Gamma^+$.

- **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.

\[\Box\]
Lemma 22 (Type preservation for LHS values). If $\Gamma^+; C \vdash l : d \ t$ and $l \xrightarrow{l'}$ then $\Gamma^+; C \vdash l' : d \ t$

Proof. This follows directly from Lemma 21 because any $l$ is also an $e$. \qed

Lemma 23 (Type preservation for parallel statements). If $\Delta; \Gamma^+; C \vdash P$ and $(D_i); C \vdash P \xrightarrow{l} (P_j')$, $(D_k)$ then $\{\Delta; \Gamma^+; C \vdash P_j'\}$ and $\Delta \vdash (D_k) : \Delta'$

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

- **Case Module Instantiation:** From E-Mod, we know that expanding $P$ gives a sequence of parallel statements composed of a single parallel statement $m'()()$ and a set of modules $\langle\text{module } m'() \rangle \langle d_j\ t_j\{[x_i \mapsto v_i]\}\ s_j \rangle$ $\langle t_q\{[x_i \mapsto v_i]\}\ s_q \rangle \bigcup\bigcup_{k}^{P_r} (D_h)_{k\in H(k)}$. First we prove that $m'$ as defined above is well-typed under $\Delta$. From T-Body, we require that $d_j \neq \phi$ and that all parallel statements are well-typed under the following environment: $\langle s_j : d_j\ \text{order}(t_j\{[x_i \mapsto v_i]\}\))\rangle \langle s_q : \{\text{in, out} : \text{order}(t_q\{[x_i \mapsto v_i]\})\}$. The first part is true because $d_j$ are the exactly the same as they were in $m$ which is itself well-typed since $P$ is well-typed. The second part is true because it follows from along with the fact that all other modules $\bigcup_{k}^{P_h} (D_h)_{k\in H(k)}$ are well-typed but this under one condition: all $p_k\{[x_i \mapsto v_i]\}$ must be well-typed under the environment we just described which is precisely equal to $\langle s_j : d_j\ \text{order}(t_j)\rangle \langle s_q : \{\text{in, out} : \text{order}(t_q)\}\rangle\{[x_i \mapsto v_i]\}$. This can clearly be obtained by applying Lemma 16. Finally, we want to prove
that $\Delta; \Gamma^+; C \vdash m'(\langle l'_j \rangle)$. Since $P$ is well-typed, we know from T-Mod that all expressions $e_i$ and $l_j$ are well-typed. We also know that $m$ is well-typed and that the widths of each $l_j$ is equal to the width of the corresponding $t_j\{[x_i \mapsto e_i]\}$. Using E-Mod, Lemma 22, Lemma 17, and T-Mod this can be obtained directly.

- **Case Assign**: From E-Assign, we know that expanding $P$ gives a sequence of new parallel statements composed of one parallel statement $P'$ equal to assign $l' e'$ and an empty set of modules. From T-Assign we know that both $l$ and $e$ are well-typed under $\Gamma^+$ and using Lemmas 22 and 21 we can conclude that $l'$ and $e'$ have the same types under $\Gamma^+$ therefore using T-Assign again we reach $\Delta; \Gamma^+; C \vdash assign l' e'$. 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 expanding $P$ gives a sequence of new parallel statements $\bigcup_k (P_k)^{\forall e \in R(k)}$, and a sequence of modules $\bigcup_k (D_h)^{\forall e \in H(k)}$. From T-If, we know that $\{\Delta; \Gamma^+; C, e \vdash P_k\}$ 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 expansion of each of them are well-typed.

  - **Case False**: Similarly we can show that all parallel statements and mod-
ules generated from the expansion 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 expanding \( P \) 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 expanding \( P \) gives a sequence of new parallel statements \( \bigcup_k \langle P_r \rangle_{r \in R(k)} \cup \langle P_j \rangle \), and a sequence of modules \( \bigcup_k \langle D_h \rangle_{h \in H(k)} \cup \langle D_q \rangle \). These sequences are generated from expanding the body of the for loop after substitution and expanding another for statement. First we know from T-For and Lemma 20 that \( \Gamma^+; C \vdash^0 v_1, v_2 : \{\text{in}\}\text{int} \). we also know that \( \{\Delta; \Gamma^+, y : \{\text{in}\}\text{int}; C, y \geq e_1, y < e_2 \vdash P_k \} \), that \( \Gamma^+, y : \{\text{in}\}\text{int}; C \vdash^0 e_3 : \{\text{in}\}\text{int} \) and that \( C \triangleright e_3 > 0 \). Using Lemmas 16 and 18 respectively we get \( \{\Delta; \Gamma^+[y \mapsto v_1]; C, y \geq e_1, y < e_2 \vdash P_k[y \mapsto v_1] \} \) and that \( \Gamma^+[y \mapsto v_1]; C \vdash^0, e_3[y \mapsto v_1] : \{\text{in}\}\text{int} \). But given that \( y \) was local to the loop, we know that none of the variables in \( \Gamma^+ \) has a type involving \( y \). Therefore \( \Gamma^+[y \mapsto v_1] = \Gamma^+ \) and we now know that \( \{\Delta; \Gamma^+; C, y \geq e_1, y < e_2 \vdash P_k[y \mapsto v_1] \} \) and that \( \Gamma^+; C \vdash^0, e_3[y \mapsto v_1] : \{\text{in}\}\text{int} \). Using the induction hypothesis we get
∀k.Δ; Γ^+; C, y ≥ e_1, y < e_2 ⊢ ⟨P_r⟩^{e \in R(k)} and Δ ⊩ \bigcup_k ⟨D_h⟩^{h \in H(k)} : Δ_1.

Finally using T-For again we get that \{Δ; Γ^+; C ⊢ for(y = v_1 + e_3[y \mapsto v_1]; y < e_2; y = y + e_3) P_k\} which by the induction hypothesis shows that ∀j.Δ; Γ^+; C ⊢ P_j and Δ; C ⊢ ⟨D_q⟩ : Δ_2 Since all the resulting parallel statements and modules are well-typed under Δ and Γ^+ then we have the required result.

□

**Lemma 24** (Type preservation for main module body). If Δ ⊩ b : M and Δ ⊩ b \overset{1}{\mapsto} b', ⟨D_k⟩ then Δ ⊩ b' : M and Δ ⊩ ⟨D_k⟩ : Δ'

**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 \{⟨d_j t_j s_j⟩ is ⟨t_q s_q⟩ \bigcup_k ⟨P_r⟩^{e \in R(k)} and a set of modules \bigcup_k ⟨D_h⟩^{h \in H(k)} \}. We also know from T-Body that each parallel statement P in P_k is typed under Δ and ⟨s_j : d_j order(t_j) ∪ s_k : \{in, out\} order(t_k)⟩, therefore using Lemma 23 we can conclude that \bigcup_k ⟨P_r⟩^{e \in R(k)} and \bigcup_k ⟨D_h⟩^{h \in H(k)} are well-typed proving the required result. □

We are now ready to prove the Type Preservation Theorem (Theorem 4). The proof works as follows:

**Proof.** Let p = ⟨D_l⟩m. From E-Prog, p' is ⟨D_r⟩ ∪ ⟨module_m b'⟩m. We also know that Δ ⊩ b \overset{1}{\mapsto} b', ⟨D_r⟩ where b is the body of the main module of p. From T-Prog,
we know that for $p$ to be typed, all $(D_i)$ 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 proceeding it. This applies to the main module too. Therefore we can conclude that $\Delta \vdash b : M$. Given that and using Lemma 24 we can conclude that $\Delta \vdash b' : M$ and $\Delta \vdash (D_r) : \Delta'$ which proves that $\vdash p'$

D.2.3 Type Safety

Proving Type Safety Theorem (Theorem 5) is trivial.

Proof. Since $\vdash p$ and $p \xrightarrow{1} p'$, we know from Theorem 4 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 $p' \neq \text{err}$

D.2.4 Preprocessing Soundness

Lemmas 25, 26, 27, 28, and 29 establish that the result of preprocessing $e$ (2 lemmas), $I$, $P$, and $b$ terms respectively produces a fully expanded term of the same category. Theorem 6 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 $\tilde{X}$

Lemma 25 (Preprocessing soundness for level 0 expressions). If $e \xrightarrow{0} e'$ then $e' \in \tilde{e}$

Proof. We proceed by induction on the evaluation tree of $e$. E-Int0 case is immediate since $e' = v$ which is allowed in $\tilde{e}$. 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 \( \hat{e} \).

\( \square \)

**Lemma 26** (Preprocessing soundness for level 1 expressions). If \( e \overset{1}{\rightarrow} e' \) then \( e' \in \hat{e} \)

*Proof.* We proceed by induction on the evaluation tree of \( e \). E-Id1 case is immediate since \( e' = s \) which is allowed in \( \hat{e} \). Cases E-Index1, E-Range1 make use of Lemma 25 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. \( \square \)

**Lemma 27** (Preprocessing soundness for LHS values). If \( l \overset{1}{\rightarrow} l' \) then \( l' \in \hat{l} \)

*Proof.* Follows directly from Lemma 26 \( \square \)

**Lemma 28** (Preprocessing soundness for parallel statements). If \( \langle D_i \rangle \vdash P \overset{1}{\rightarrow} \langle P'_j \rangle, \langle D_k \rangle \) 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 \( m'(\langle l \rangle) \) and a set of modules \( \langle \text{module } m' \rangle \langle d_j t_j \{ [x_i \rightarrow v_i] \} \rangle s_j \) is \( \langle t_j \{ [x_i \rightarrow v_i] \} \rangle \langle \bigcup \langle P_t \rangle_{t \in R(k)} \rangle \cup \langle \bigcup \langle D_h \rangle_{h \in H(k)} \rangle \) \( P' \) 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 \( P_k \{ [x_i \rightarrow v_i] \} \) 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 \( v_\ell \) is in \( \hat{e} \) from Lemma 25.

- **Case Assign**: From E-Assign, we know that expanding \( P \) gives a sequence of parallel statements composed of one parallel statement \( P' \) equal to assign \( l' e' \) and an empty set of modules. Using Lemmas 27 and 26 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-False, The expansion result is equal to an empty sequence of parallel statements and an empty sequence of modules.

- **Case True**: From E-True, 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-True 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 $\hat{\hat{e}}$ as proved in Lemma 25.

\[\square\]

**Lemma 29** (Preprocessing soundness for main module body). If \(\langle D_\ell \rangle \vdash b \rightarrow b', \langle D_k \rangle\) 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 $\bigcup_j \langle d_j t_j s_j \rangle$ is $\langle t_q s_q \rangle \bigcup \langle P_r \rangle_{r \in R(k)}$ and a set of modules $\bigcup_k \langle D_h \rangle_{h \in H(k)}$. Note that $b'$ is parameter free and that from Lemma 28 we know that all parallel statements used to construct $\bigcup_k \langle P_r \rangle_{r \in R(k)}$ are in $\hat{P}$ and that
all module definitions used to construct $\bigcup_k (D_h)^{h \in H(k)}$ 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. \qed

Finally, we prove the Preprocessing Soundness Theorem (Theorem 6) as follows:

Proof. From E-Prog, $p'$ is $(D_r) \cup (\text{module } m \ b') m$ where $(D_h) \vdash b \leftarrow b', (D_r)$ and $b$ is the body of the main module of $p$. From Lemma 29 we know that for every $D_r$, we have $D_r \in \hat{D}$ and that $b' \in \hat{b}$, therefore $p'$ is in $\hat{p}$. \qed