

Hardware descriptions as two-level computations

Walid Taha, Yousra Alkabani,
Cherif Andraos, Jennifer Gillenwater,
Gregory Malecha, Angela Yun Zhu
Rice University, Houston, TX, USA
taha@rice.edu

Jim Grundy, John O'Leary
Intel Strategic CAD Labs
Portland, OR, USA
jim_grundy@ichips.intel.com

Both software engineers and digital circuit designers face the fundamental challenge of managing increasingly complex computational artifacts that must function with exacting precision. The authors have recently embarked on a joint project aimed at exploring how programming language innovations can help hardware designers. In particular, the project explores how statically typed two-level languages (STTL) can be used to improve hardware description languages. This abstract is a brief overview of the project and results to date.

Static typing is a verification technology that has proven to scale linearly in practice. Advanced static type systems that support indexed types have been used to perform static array bound checking. We expect that the same idea can be used to check the consistency of the width of various buses in a digital circuit. In addition, we will explore their use for checking the size and shape of circuits. Two-level languages [2] are a formal model for programming languages with a notion of a preprocessing stage. They have been used to formalize the internal workings of offline partial evaluators [1], to write statically-checked programming generators [4], and as a basis for designing statically typed macros [5].

In the software world, STTLs have been successfully used to express generic, highly parameterized designs that are both machine-checkable and highly modular. The notion of static typing for an STTL program is strong: the static guarantee is that the *result of preprocessing* will be well-typed. Can STTLs do for digital circuit designs what they have done for software?

Several similarities between software engineering and hardware design suggest that STTLs can help. High-performance programs often completely avoid the use of powerful abstraction mechanisms because of their associated runtime cost. Two-level languages allow us to write programs where uses of abstractions such as functions, objects, and modules are eliminated at compile time. Similarly, hardware designers working on high-performance circuits are required to write carefully stylized hardware descriptions to ensure these descriptions are indeed synthesizable by the tool being used. This similarity suggests two approaches: The first approach is to extend existing hardware description languages with STTL features that can express designs in a more modular fashion, and that are guaranteed to expand into well-formed descriptions that meet the traditional stylistic constraints that hardware designers must comply with. Work on hardware description languages in the functional programming community shows that adding a mostly-functional language for first-stage (or expansion-time computation) can be a highly attractive approach. But a closer analysis of mainstream hardware description languages suggests a second approach. In particular, Verilog and VHDL have constructs (like `for` and `while` statements) that are eliminated in a first stage. Can type systems for two-level languages be used to specify precisely the conditions under which such constructs can be eliminated in an preprocessing stage? If we can do that, we would have a much less invasive approach to migrating existing engineering practice to a point where higher level descriptions expressible in existing hardware descrip-

tion languages can be used without jeopardizing the fidelity of the design process.

As a first step in this direction, we show how two-level languages can be used to define a two-level type system that characterizes Verilog programs guaranteed to expand to ones that are “obviously synthesizable”. Viewing synthesizable descriptions as two-level programs can have several benefits. It provides the terminology for explaining the behavior of any particular synthesis tool, and for expressing the precise behavior that a hardware designer expects from a synthesis tool. In addition, it makes it possible to compare techniques for binding-time improvement in partial evaluators to synthesis methods used in hardware synthesis tools.

So far, we have defined the syntax for a core calculus (a representative subset of Verilog) and defined a first-stage semantics to model preprocessing, as well as a static type system that checks that a given program is synthesizable. Key properties demonstrating the consistency of this model are presented. First, a well-typed two-level program synthesizes a well-typed one-level Verilog program. Second, the synthesized program is free of preprocessing (or “stage one”) computations. Third, preprocessing is safe in the sense that preprocessing computations do not depend on wire values.

To validate the core calculus developed in this work, we implemented a prototype implementation that supports the core subset using Verilog-compatible concrete syntax. The prototype will be made available online.

A more detailed write up of this is currently underway. In addition to reporting on the technical results, we expand on the more subtle lessons that we have learned so far from this work, including how to treat the notion of synthesizability as well as the similarities between problems that arise in software program generation and hardware synthesis.

Acknowledgments: We would like to thank Scott Rixner and Kartik Mohanram for interesting and stimulating discussions on the goals of this work.

1. REFERENCES

- [1] Carsten K. Gomard and Neil D. Jones. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.
- [2] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, 1988.
- [3] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000, USA. Available online from <ftp://cse.ogi.edu/pub/tech-reports/README.html>.
- [4] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [3].
- [5] Walid Taha and Patricia Johann. Staged notational definitions. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

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