Π-Ware: An Embedded Hardware Description Language using Dependent Types

João Paulo Pizani Flor and Wouter Swierstra

Dept. of Inform. and Comput. Sci., Universiteit Utrecht, The Netherlands
{j.p.pizaniflor|w.s.swierstra}@uu.nl

Computer hardware has experienced a steady exponential increase in complexity in the last decades. There is an increased demand for application-specific integrated circuits that avoid the proverbial *von Neumann bottleneck* [1], and ever more algorithms enjoy hardware acceleration (such as 3D/2D renderers, video/audio codecs, cryptographic primitives and network protocols).

This demand puts pressure on the industry to make hardware design quicker and more efficient. At the same time, hardware design also requires very strong correctness guarantees. These strong correctness requirements are commonly met using (exhaustive) testing and model checking, making the design-verify-fix loop slower and more costly than in the software world.

There is a long tradition [4, 6] of modeling hardware using functional programming to pursue higher productivity and stronger correctness assurance. A particular trend is to use functional programming languages as *hosts* for Embedded Domain-Specific Languages (EDSLs) aimed at hardware description, of which Lava [2] is a popular example. Some of the limitations of these hardware EDSLs are due to the host's type system, which lacks the power to express some desirable properties of circuit models.

We believe that the advantages brought to hardware design by FP-inspired techniques can be even greater if we use dependent types. We define a Hardware Description Language (HDL) called Π-Ware, which is *embedded* in the dependently-typed general-purpose programming language Agda. Circuits described in Π-Ware can be simulated, transformed in several ways, proven correct, and synthesized (work in progress).

The existence of several different semantics for circuit models is due to Π-Ware's *deep embedding*: There is an inductive datatype of circuits (called C), and semantics (simulation, synthesis, gate count, etc.) are just functions with C as domain. More specifically, the circuit datatype is *indexed* by two natural numbers representing, respectively, the *sizes* of the circuit's input and output. The arithmetic fine tuning of these indices, along with other features of dependent types, allow us to "ban" certain classes of design mistakes *by construction*, for example:

- Our circuit constructors guarantee that all circuits are *well-sized*, i.e., there are never "floating" or "dangling" wires.
- Connections between circuits are guaranteed to never cause *short-circuits* (two or more sources connected to the same load).

In addition to this *structural correctness by construction*, we can also *interactively* prove properties of circuits, which provides some advantages over the fully-automated verification approach used widely in industry. First of all, we can inductively prove properties over whole *circuit families*. Furthermore, proofs written in Agda are *modular*, in contrast to fully-automated verification. This means that, when proving laws concerning a certain circuit (family), we *reuse* previously proven facts about its constituent subcircuits.

In particular, we can write proofs of *functional correctness*. Our simulation semantics (even for sequential circuits) are *executable* (in contrast to other EDSLs [3] with a *relational* semantics). This means that the simulation semantics simply converts each circuit into a function,

which can then be applied to input vectors, producing output vectors. We can then formulate the statement of whether a circuit (family) *implements* the behaviour of a given *specification function*, and proofs of correctness can be written by induction on circuit inputs.

As a first step in exploring the possibilities that our approach offers, we conducted a case study, aiming to formalize a class of circuits known as *parallel-prefix sums* [5] in Π-Ware. Parallel-prefix circuit combinators have been defined *in terms of* Π-*Ware primitives*, and we are proving several properties of this class of circuits which had previously only been postulated or proven on paper.

The case study has led us to establish suitable *equivalence relations* between circuits. We have defined a relation of *equality up to simulation*, which identifies any two circuits with the same simulation behaviour (taking equal inputs to equal outputs). Reasoning about this equivalence relation, without postulating function extensionality, presents several challenges, for which we believe to have found satisfactory solutions. Finally, we show how to define *algebraic* laws involving circuit constructors and combinators. These may be used to construct *provably safe* circuit transformations.

All in all, we claim there are several benefits to be gained from using dependent types to describe hardware circuits. These techniques are more widely applicable to other domains and, therefore, we believe them to be of interest to the wider TYPES community.

References

[1] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. of ACM*, 21(8):613–641, 1978.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. *SIGPLAN Not.*, 34(1):174–184, 1998.

[3] T. Braibant. Coquet: a Coq library for verifying hardware. In J.-P. Jouannaud and Z. Shao, eds., *Proc. of CPP 2011*, v. 7086 in *Lect. Notes in Comput. Sci.*, pp. 330–345. Springer, 2011.

[4] P. Gammie. Synchronous digital circuits as functional programs. *ACM Comput. Surv.*, 46(2):21, 2013.

[5] R. Hinze. An algebra of scans. In D. Kozen, ed., *Proc. of MPC 2004*, v. 3125 in *Lect. Notes in Comput. Sci.*, pp. 186–210. Springer, 2004.

[6] M. Sheeran. Hardware design and functional programming: a perfect match. *J. of UCS*, 11(7):1135–1158, 2005.