6. Research Proposal

6.a Description of the proposed research

Computer hardware is becoming increasingly complex. Nowadays most personal computers are furnished with several cores sharing gigabytes of memory behind multi-level caches.

As a result of this increasing complexity, the cost of the formal verification of new designs is rapidly rising. Verification now consumes more than two thirds of all development resources associated with new hardware designs. Skimping on verification is not an option: the notorious error in the floating point unit of a Pentium processor is estimated to have cost 500,000,000 dollars [18].

Traditionally, most hardware designs are checked through simulation. Once a circuit successfully passes a modest number of test cases and produces the correct result, it is simulated for an extensive period of time—passing different inputs to the circuit to check that it satisfies its specification. It is not uncommon for such simulations to run for several months on mainframe computers. When such simulations do uncover a bug, the design team must fix the problem and start another round of simulation to validate the fix.

The cost of simulation has motivated research into techniques to find design errors more quickly—model checking [5, 17] and symbolic simulation [11, 12] being two popular approaches. Yet even using such advanced techniques, there is still a clear divide between the design and verification process. Verification engineers are often still required to reconstruct the designers' intent. To make matters worse, uncovering an error may require a complete new series of simulations. To reduce the overall cost of verification, it is essential to tightly couple the design and verification process [31, 50].

Aims and objectives: This proposal aims to develop a domain-specific language both to describe and to verify formally circuit designs, thereby integrating the design and verification process. To do so, we will explore how to *raise the level of abstraction* available in the current generation of hardware description languages by applying ideas from the programming language research community: higher-order functional programs written in a constructive higher-order logic.

To this end, this proposal sets the following specific objectives:

1. To explore the use of a general purpose dependently-typed programming language to host a domain-specific language for the description of hardware.

The circuits written in the proposed language will be executable as well as generate VHDL designs.

2. To enforce static invariants of circuits using dependent types, thereby already precluding certain errors during the design process;
3. To prove that circuit designs meet their specification and to identify the proof combinators that facilitate the construction of reusable proofs;
4. To develop a notion of *refinement* between specifications and low-level circuit descriptions, enabling the incremental development of formally verified circuits.

Upon its completion, this project will provide hardware designers with a means to catch errors early in the development cycle and to convert verification effort into reusable proofs. This research we propose is not a silver bullet to solve all the problems associated with hardware verification, but rather a new and exciting answer to an important problem.

Background and motivation

Functional programming and hardware design The current generation of hardware description languages, such as Verilog and VHDL, has been designed to simulate circuit descriptions. Unfortunately, verification based exclusively on simulation is no longer viable. Rather than adapt existing languages, this proposal seeks to exploit exciting developments in program language design. In particular, we will draw from ideas relating hardware design and functional programming languages [52]. Languages such as Intel's Forte [25] demonstrate that functional languages are already influencing existing hardware description languages.

One promising approach to designing a new hardware description language is Lava [7], a domain-specific language for hardware, embedded in the general-purpose functional language Haskell [46]. Instead of implementing a new hardware description language from scratch, Lava consists of a library of Haskell functions to define circuits. This approach, embedding a domain-specific language within a general purpose functional language, has been applied to many different domains including financial contracts [47], parsing [33], pretty-printing [30], randomised testing [14], music descriptions [29], and animation [21]. Lava itself has been used in industry with great success at Xilinx by Singh [34, 54].

At the lowest level, Lava provides several functions to describe individual gates. More interesting, are Lava's *circuit combinators*—functions that assemble large circuits from several smaller components. For instance, you might want to compose two circuits sequentially or in parallel. Besides such simple compositions, Lava provides other combinators that describe rich circuit designs, such as tree-shaped circuits and butterfly circuits. Furthermore, circuit designers can use Haskell's abstractions to define their own combinators to express any new patterns as they emerge in their design.

A Lava circuit design may serve different purposes. Besides extracting a VHDL description of the circuit, there are several verification techniques Lava supports. As the description is executable, you can test Lava circuits with QuickCheck [14], providing a light-weight correctness check. Secondly, it is possible to symbolically evaluate Lava circuits. Finally, Lava circuits may generate proof obligations for automatic theorem provers.

All these verification techniques are fully automated and require no user interaction. While this may seem appealing, the class of properties for which an automatic decision procedures exist is restricted and randomised testing can provide only limited guarantees. Therefore other approaches to hardware verification use theorem provers [8, 24, 38], most notably Isabelle/HOL [43] and ACL2 [10, 32], to establish formal properties of a circuit's specification. Such interactive theorem provers are already being used to great effect by the hardware verification community, as witnessed by Fox's verification of the ARM6 microprocessor [22] or Harrison's work on floating point verification at Intel [27].

While verification with such interactive proof assistants requires more work by hand than fully automatic provers, there are no restrictions on the properties that can be formalised thusly. This approach does have its drawbacks. Most notably, there is still a large gap between the hardware design and verification effort. The verification is usually done on a mathematical model of the circuit, and not on the circuit description itself. It does not necessarily guarantee that the actual circuit has been implemented in accordance with the verified mathematical model—a limitation we will strive to overcome.

Since the design of Lava, there have been new developments in the design of functional programming languages with *dependent types*—which may be used to integrate Lava with interactive theorem provers.

Dependent types Computer programs manipulate data. A *type system* prevents a program from using data in the wrong way. A program fragment such as

$$\mathit{isItSaturdayToday} \div \texttt{"Hello"}$$

does not make sense. Division is only defined on numbers. Trying to divide anything by a piece of text indicates a program error. A program that is *type-correct* is guaranteed not to make such errors.

Now consider the following two deduction rules:

$$\frac{\mathit{isEven} : \mathit{Int} \to \mathit{Bool} \quad 5 : \mathit{Int}}{\mathit{isEven}\ (5) : \mathit{Bool}} \qquad \frac{p \to q \quad p}{q}$$

The deduction on the left may be part of the type checking performed before a program is executed; on the right, is the logical *modus ponens* rule. Note that, when you only consider the types to the right of the colon, these two rules have exactly the same structure. This correspondence, known as the Curry-Howard isomorphism [55], states that every type system may be viewed as a system of formal logic and vice versa.

The type system of most conventional programming languages corresponds to (some variation of) propositional logic. Hence, types cannot carry a great deal of information about the values that inhabit them. It is impossible to write down the *type* of all non-zero binary words of width *n*; or the type of an *n* by *m* matrix; or the type of a 16-bit address greater than 0x8hc2. All these examples constitute *types depending on values*, or so called *dependent types*.

Dependent types were originally developed by Martin-Löf as a formal foundation of constructive mathematics. Many proof assistants based on dependent types, such as Automath [20] or Coq [6], have already been used to formalise significant parts of mathematics.

Starting with Augustsson's Cayenne [3] language eleven years ago, there has been an increased interest in dependently typed *programming languages*. In particular, McBride and McKinna's work on Epigram [36, 37] has shown how to implement a programming language on top of a consistent type theory. Many of the ideas underlying the Epigram prototype have made their way into Agda [44], a programming language developed by Norell with a more scalable implementation. Coq is gradually supporting more programming facilities [56]. These systems have been used to implement verified parsers [13], data structures [42], and database servers [60].

Research programme

The key idea underlying this proposal is to embed a domain-specific language for hardware description in a functional language with dependent types. This idea has been proposed before [9, 26], but it only recently that implementations of programming languages with dependent types have matured enough to explore the full potential of this approach[44, 57, 56]. This is the perfect time to undertake this research.

Note that the aim is not to replace existing hardware description languages, such as VHDL or Verilog, but rather to leverage new programming language technology to address existing problems in this domain. To ensure the research can be used together with existing circuit designs, it will be possible to extract VHDL descriptions and execute circuit designs.

More specifically, this proposal will investigate how dependent types can *catch design errors before verification has even started*; examine how *complex correctness proofs can be assembled modularly* from a handful of proof combinators; and finally, develop a notion of *refinement* between circuit specifications and their complete implementations.

1 Static invariants Dependent types afford programmers with the opportunity to express a function's invariants in its type. For example, you may want to ensure that the insertion and deletion operators of an AVL tree respect the balancing factor appropriately. By encoding such invariants in the types of your functions, the type system will guarantee that they are never violated.

In the context of hardware description languages, there are several invariants that simple type systems cannot express. For example, Bluespec, Inc. have designed a propriety high-level hardware description language [41]. One of the key differences between their type system and Haskell is their use of *numeric types* to enforce bit-width and size constraints. Similarly, you may want to bound a sequential circuit's delay—ensuring it produces a value within at most k clock cycles. Such invariants are much harder to enforce in languages or proof assistants without dependent types.

The proposed research project aims to *identify light-weight invariants important to hardware designers and to enforce these invariants statically using the type system*. As a result of this work, *circuit designers will be confronted with errors earlier on in the design process, before verification has even started*. This has the potential to reduce drastically the number of simulation cycles necessary.

2 Proof combinators A language with dependent types provides a unified framework for proofs and programs. It is possible to write machine-checked proofs that a circuit matches its specification, but such formal proofs can become large and unwieldy.

Using Lava, many complex hardware designs can be expressed by composing smaller circuits: a binary adder circuit can be implemented by wiring together a series of full adder circuits that each add two bits. Unfortunately, Lava does not exploit this structure during verification. The challenge is to find a suitable set of *proof combinators* to make large proofs manageable.

The second component of this research project consists of *engineering a library of proof combinators that facilitate the verification of composite circuits*. This work will *enable the compositional verification of circuit designs*—to prove a complex circuit satisfies its specification, it will suffice to verify the circuit's subcomponents separately and construct a larger proof using the proposed proof combinators. One example of such a proof combinators is a dependently typed view [37, 59], or custom induction principle. Such higher-order properties are not easy to express in proof assistants with a more restricted logic such as ACL2.

Developing a library of verified components in this fashion can cut back the overall cost of circuit verification. Finally, these combinators will make it possible to prove the correctness of *parametrised circuit generators* [51] and circuits with unbounded state spaces—two domains where many traditional verification techniques struggle.

3 Refinement The previous section discusses how to verify that a circuit description meets its specification. It does not say anything about how to *derive* the circuit definition starting from a specification.

Yet this is an important question to ask: when designing circuits you often want to start with a high-level specification, describing the expected behaviour of a microprocessor, for instance. From this specification, designers may start implementing parts of the design all the way down to its individual gates; or alternatively, designers may want to optimize this design, adding pipelines and caches. Either way, the circuit should continue to meet its original specification—only more and more implementation details and optimizations are added.

A final aim of this project is to adapt ideas from the *refinement calculus* [4, 39], a logic of verified software, to develop a similar logic of circuit refinement. This would make enable us to unify our previous work on high-level hardware specification languages [15, 58] and low-level Lava circuit descriptions—refinement relation describes the steps from a mathematical specification all the way down to a circuit's gate-level implementation. Although there have been previous notions of refinement for specific problems in hardware design [2, 35], we propose to tackle this problem in its full generality.

This part of the proposed research has the potential to make a significant impact: programming languages with dependent types are by their very nature designed to provide single language of programs, specifications, and proofs—and establishing the relation between these concepts is precisely where many existing hardware description languages fall short.

Research embedding The research will be carried out at the Radboud University of Nijmegen by dr. Swierstra and forms a natural successor to dr. Swierstra's current NWO Rubicon project, *The Logic of Interaction*. The Foundations research group in Nijmegen is involved with several joint European efforts on type theory and mathematics, such as the EU STREP project ForMath and the EU Coordination Action TYPES. There is further local expertise on hardware verification in the form of dr. Schmaltz and his NWO EviDAM project. In summary, the Radboud University is one of the few universities combining strong research in both hardware verification and type theory.

6.b Application perspective

The current generation of hardware description languages are struggling to cope with the skyrocketing cost of the verification of modern circuits. The recent International Technology Roadmap for Semiconductors [1] states that:

> *without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry.*

The proposed research addresses this problem.

This research will replace the current standard tools, such as Verilog or VHDL, in the immediate future. Hardware manufacturers have invested too much in the existing technology to justify the adoption of new, experimental hardware description languages. In the longer run, however, experimental domain-specific languages such as the one proposed here may influence the next generation of hardware description languages or may be used to complement existing verification tools.

Nowadays there are domain-specific languages establishing a foothold in the hardware industry. A good example of this is Cryptol [23], a domain-specific language for cryptographic algorithms developed by Galois, Inc. Algorithm descriptions in Cryptol are compiled to different back-ends, such as VHDL or C. The Cryptol descriptions of crypto-algorithms tend to be much easier to write, reason about, and verify. In previous work, dr. Swierstra has already shown how key features of the Cryptol language can be implemented in a language with dependent types [45]. In certain domains, like cryptography, where correctness is critical, the potential reduction of verification cost already outweighs the price associated with the adoption of novel technology.

Beneficiaries

The proposed research has several beneficiaries from different communities:

Hardware designers The hardware language community will be one of the main beneficiaries of the proposed research. The proposed research aims to transfer technology from programming language design and type theory to the hardware community.

Functional programmers and type theorists There is a growing interest into programming with dependent types. Functional language designers are gradually developing new language features that encode more information in a value's type [49, 48]. On the other end of the spectrum, type theorists are exploring how to apply their ideas in functional languages [36]. One of the missing ingredients here is large case studies. To bring these two research communities (functional programmers and type theorists) closer together, it is necessary to establish further common ground: practical applications of dependent types that appeal to functional programmers and type theorists alike. The proposed research fills this niche.

Program language designers Functional languages, such as Haskell, are starting to have a significant impact on mainstream languages. Many of the recent developments in Java and C#, such as delegates and generics [40], are inspired by technology from the functional programming community. Language architects at leading companies, such as Erik Meijer (Microsoft) or Guy Steele (Sun), have a background in functional programming. F#, a functional language developed at Microsoft, is now part of the .NET framework and rapidly gaining popularity.

An auxiliary aim of this research proposal is to continue pushing the boundaries of what is possible using functional languages. In the long run, the results of this research will help shape the future developments of mainstream languages.

Industrial contacts

We have several contacts with companies where this research can be applied. Dr.

Swierstra's previous postdoctoral research was funded by Intel, with Carl Seger being his main contact. We fully expect to continue this collaboration when undertaking this research.

Nationally, dr. Swierstra was previously employed by Vector Fabrics, a high-tech startup specialised in applying functional languages in the development of embedded systems. Although the results of this research will not be of immediate corporate value, it will be interesting to investigate how the proposed domain-specific language may serve as a specification language for their customers.

7. Project planning

We request funding for a three year postdoc position. We have identified separate Work Packages (WP) and scheduled them in the Gantt chart below (Figure 1).

Year 1 In the first year, the research will focus on *combinational circuits*, that is, circuits whose outputs only depend on their inputs. We do not foresee any technical difficulties executing the proposed research for this class of circuits as they correspond to boolean logic formulae, for which (semi)automatic solvers exist.

At the end of the first year, we will have a prototype language design and implementation in the programming language and proof assistant Coq [6], made publicly available under an open source license (WP1). It will be possible to (symbolically) simulate these circuit descriptions and extract corresponding VHDL from

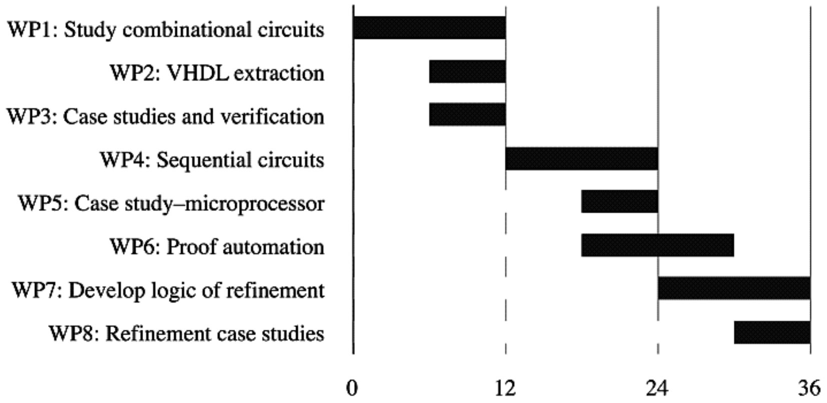

Figure 1: Gantt diagram for proposed research

them (WP2). We expect to publish our initial results within the first year in order to advertise this project to the wider community, and collect feedback from other experts.

To evaluate our results, we will implement and verify several small, classic examples of Lava circuit designs such as (prefix) adders [53] or sorter cores [16] (WP3). Our aim is not to design exciting new hardware, but rather to compare our approach to existing technology and, where necessary, refine our design choices at this early stadium.

Year 2 We will show how these initial results can be extended to *sequential circuits*, that is, circuits that may maintain some internal state, in the second year (WP4). The main challenge will be to deal with feedback loops and develop a suitable logic for reasoning about such loops. Recent results on mixed inductive and coinductive data type definitions may provide a welcome foothold [19].

At the end of the second year, we will have several more substantial examples of circuit descriptions and verification developments. One typical case study would be the formal verification of a simple microprocessor as described by Hennessy and Patterson [28] (WP5). To ensure the correctness proofs scale along with our circuit descriptions, it is important to identify high-level proof principles and opportunities for automation (WP6). Once again, it will be possible to simulate the circuits described in our case studies and extract VHDL from them.

Year 3 In the final year, we will develop a notion of *refinement* between circuits and their specifications (WP7). Drawing on existing work on refinement calculus [4, 39] and our work on specification languages [15, 58], we aim to show how to refine a mathematical circuit specification to a gate-level description of a hardware circuit. We will re-use the case studies developed in previous years to evaluate our results. To raise the bar even further, we will establish how certain circuit optimizations, such as pipelines and memory caches, can also be considered refinements of a (more naïve) circuit with the same behaviour (WP8).

8. Expected use of instrumentation

The proposed research requires only the use of stock laptop computers and open source software.

Word count: Sections 6, 7, and 8 use a total of 3494 words.

Five important publications

The five most important publications of the research team relevant to this proposal are:

- Per Bjesse, Koen Claessen, Mary Sheeran and Satnam Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, 1998.
- Herman Geuvers and Henk Barendregt. Proof-assistants using dependent type systems. *Handbook of automated reasoning*, Volume 2. 2001.
- Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
- Nicolas Oury and Wouter Swierstra. The Power of Pi. In *ICFP 08: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*, 2008.

- Wouter Swierstra, Koen Claessen, Carl Seger, Mary Sheeran, and Emily Shriver. Chalk: a language and tool for architecture design and analysis. *Eighth International Workshop on Designing Correct Circuits*, 2010.

References

[1] International Technology Roadmap for Semiconductors. Available from http://www.itrs.net., 2007.

[2] J.R. Abrial. *The B-book: assigning programs to meanings*. Cambridge Univ Press, 1996.

[3] Lennart Augustsson. Cayenne—a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250. ACM, 1998.

[4] R.J. Back and J. Wright. *Refinement calculus: a systematic introduction*. Springer Verlag, 1998.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program De-*

velopment. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[7] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, 1998.

[8] Peter Böhm and Tom Melham. A refinement approach to design and verification of on-chip communication protocols. In *Proceedings of the Eighth Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*. IEEE Computer Society, 2008.

[9] Edwin Brady, James McKinna, and Kevin Hammond. Constructing correct

circuits: Verification of functional aspects of hardware specifications with dependent types. In *Trends in Functional Programming*, pages 159–176. Intellect Ltd., 2008.

[10] Bishop Brock, Matt Kaufmann, and J Moore. ACL2 Theorems about Commercial Microprocessors. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*. Springer Verlag, 1996.

[11] R.E. Bryant. Symbolic simulation—techniques and applications. In *Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 517–521. ACM New York, NY, USA, 1991.

[12] W.C. Carter, W.H. Joyner Jr, and D. Brand. Symbolic Simulation for Correct Machine Design. In *Design Automation, 1979. 16th Conference on*, pages 280–286, 1979.

[13] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, September 2009.

[14] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

[15] Koen Claessen, Carl Seger, Mary Sheeran, Emily Shriver, and Wouter Swierstra. High level architectural modelling for early estimation of power and high level architectural modelling for early estimation of power and performance. In *Hardware Design and Functional Languages*, 2009.

[16] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, Lecture Notes in Computer Science. Springer Verlag, 2001.

[17] E. M. Clarke, D. E. Long, and K. L. Mcmillan. *Model Checking*. MIT Press, 1999.

[18] Edmund M. Clarke and Robert P. Kurshan. Computer-aided verification. *IEEE Spectr.*, 33(6):61–67, 1996.

[19] Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction. Draft currently under revision., 2009.

[20] N.G. de Bruijn. *A survey of the project AUTOMATH*, pages 589–606. Kluwer Academic Press, 1980.

[21] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP '97: Proceedings of the Second ACM SIGPLAN International Conference on Functional*

Programming, 1997.

[22] A. Fox. Formal specification and verification of ARM6. *Theorem Proving in Higher Order Logics*, pages 25–40, 2003.