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

Author: João Paulo Pizani Flor
<joaopizani@uu.nl>

Supervisor: Wouter Swierstra
<w.s.swierstra@uu.nl>

Department of Information and Computing Sciences
Utrecht University

Saturday 23rd August, 2014

Universiteit Utrecht

Table of Contents

Hardware design is hard(er)

- Strict(er) correctness requirements
  - You can't simply *update* a full-custom chip after production
    - Intel `FDIV`
  - Expensive verification / validation (up to 50% of development costs)
- Low-level details (more) important
  - Layout / area
  - Power consumption / fault tolerance

Universiteit Utrecht

Hardware design is growing

- Moore's law will still apply for some time
  - We can keep packing more transistors into same silicon area
- **But** optimizations in CPUs display diminishing returns
  - Thus, more algorithms *directly* in hardware

Hardware Description Languages

- All started in the 1980s
- *De facto* industry standards: VHDL and Verilog
- Were intended for *simulation*, not modelling or synthesis
  - *Unsynthesizable* constructs
  - Widely variable tool support

Functional Programming

- Easier to *reason* about program properties
- Inherently *parallel* and *stateless* semantics
    - In contrast to imperative programming

Functional Hardware Description

- A functional program describes a circuit
- Several *functional* Hardware Description Languages (HDLs) during the 1980s
  - For example, μFP [Sheeran, 1984]
- Later, *embedded* hardware Domain-Specific Languages (DSLs)
  - For example, Lava (Haskell) [Bjesse et al., 1998]

Embedded DSLs for Hardware

- Lava
- Limitations
  - Low level types
  - Not guaranteeing size match

Dependently-Typed Programming

Dependently-Typed Programming (DTP) är en programmationstechnik...

Research Question

“What are the improvements that DTP can bring to hardware design?”

Methodology

- Develop a hardware DSL, *embedded* in a dependently-typed language (Agda)
  - Called **Π-Ware**
  - allowing simulation, synthesis and verification

Dependently-Typed Programming

- Types can depend on values
  - Example: data Vec $(\alpha : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}$ where...
  - Compare with Haskell (GADT style):
    `data List :: * -> * where`...
- Types of arguments can depend on *values of previous arguments*
  - Ensure a "safe" domain
  - $\mathsf{take} : (m : \mathbb{N}) \to \mathsf{Vec}\ \alpha\ (m + n) \to \mathsf{Vec}\ \alpha\ m$

Dependently-Typed Programming

- Type checking requires *evaluation* of functions
  - We want Vec Bool $(2 + 2)$ to unify with Vec Bool 4
- Consequence: all functions must be *total*
- Termination checker ensures (heuristics)
  - Structurally-decreasing recursion
    - This passes the check:

$$\begin{aligned}
&\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathsf{add}\ \mathsf{zero}\ \ y = y \\
&\mathsf{add}\ (\mathsf{suc}\ x')\ \ y = \mathsf{suc}\ (\mathsf{add}\ x'\ \ y)
\end{aligned}$$

    - This does not:

$$\begin{aligned}
&\mathsf{silly} : \mathbb{N} \to \mathbb{N} \\
&\mathsf{silly}\ \mathsf{zero} = \mathsf{zero} \\
&\mathsf{silly}\ (\mathsf{suc}\ n') = \mathsf{silly}\ \lfloor n' /2 \rfloor
\end{aligned}$$

Universiteit Utrecht

Dependently-Typed Programming

- Dependent pattern matching can *rule out* impossible cases
  - Classic example: *safe* head function

    $$\mathsf{head} : \mathsf{Vec}\ \alpha\ (\mathsf{suc}\ n) \to \alpha$$
    $$\mathsf{head}\ (x :: xs) = x$$

  - The **only** constructor returning $\mathsf{Vec}\ \alpha\ (\mathsf{suc}\ n)$ is $\_::\_$

Universiteit Utrecht

Depedent types as logic

- Programming language / Theorem prover
  - Types as propositions, terms as proofs [Wadler, 2014]
- Example:
  - Given the relation (drawn triangle):

$$\begin{array}{l} \mathsf{data}\ \_{\leq}\_ : \mathbb{N} \to \mathbb{N} \to \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{z{\leq}n} : \forall\ \{n\} \qquad\qquad\qquad \to \mathsf{zero} \quad \leq n \\ \quad \mathsf{s{\leq}s} : \forall\ \{m\ n\} \ \to m \leq n \ \to \mathsf{suc}\ m \ \leq \mathsf{suc}\ n \end{array}$$

  - Proposition:

$$\mathsf{twoLEQFour} : 2 \leq 4$$

  - Proof:

$$\mathsf{twoLEQFour} = \mathsf{s{\leq}s}\ (\mathsf{s{\leq}s}\ \mathsf{z{\leq}n})$$

$$\mathsf{s{\leq}s}\ (\mathsf{s{\leq}s}\ (\mathsf{z{\leq}n} : 0 \leq 4) : 1 \leq 4) : 2 \leq 4$$

Universiteit Utrecht

Agda syntax for Haskell programmers

- Liberal identifier lexing (Unicode **everywhere**)
  - a≡b+c is a valid identifer, $a \equiv b + c$ an expression
  - Actually used in Agda's standard library
  - And in Π-Ware: $\mathbb{C}$, $[\![c]\!]$, $\Downarrow$, $\Uparrow$
- *Mixfix* notation
  - _[_]≔_ is the vector update function: v [# 3] ≔ true.
  - _[_]≔_ v (# 3) true $\Longleftrightarrow$ v [# 3] ≔ true
- Almost nothing built-in
  - _+_ : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ defined in Data.Nat
  - if_then_else_ : Bool $\to \alpha \to \alpha \to \alpha$ defined in Data.Bool

Agda syntax for Haskell programmers

- Implicit arguments
  - Don't have to be passed if Agda can **guess** it
  - Syntax: $\varepsilon : \{\alpha : \mathsf{Set}\} \rightarrow \mathsf{Vec}\ \alpha\ \mathsf{zero}$
- "For all" syntax: $\forall\ n \iff (n : \_)$
  - Where _ means: guess this type (based on other args)
  - Example:
    - $\forall\ n \rightarrow \mathsf{zero} \leq n$
    - $\mathsf{data}\ \_\leq\_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathsf{Set}$
- It's common to combine both:
  - $\forall\ \{\alpha\ n\} \rightarrow \mathsf{Vec}\ \alpha\ (\mathsf{suc}\ n) \rightarrow \alpha \iff$
    $\{\alpha : \_\}\ \{n : \_\} \rightarrow \mathsf{Vec}\ \alpha\ n \rightarrow \alpha$

Universiteit Utrecht

Low-level circuits

- $\mathbb{C}'$
- "Untyped"

Atoms

- PiWare.Atom.Atomic
- Bool, `std_logic`, etc.
- Example: PiWare.Atom.Bool

Gates

- PiWare.Gates.Gates
- Examples:
  - $\{\mathtt{NOT}, \mathtt{AND}, \mathtt{OR}\}$ (BoolTrio)
  - $\{\mathtt{NAND}\}$
  - Arithmetic, Crypto, etc.
- Example: PiWare.Gates.BoolTrio

High-level circuits

- $\mathbb{C}$
- "Typed"

Synthesizable

- $\Downarrow\mathsf{W}\Uparrow$ (pronouced `Synthesizable`)
  - $\mathsf{W}\ n = \mathsf{Vec}\ \alpha\ n$
- Example: $\Downarrow\mathsf{W}\Uparrow\ (\alpha \times \beta)$

Synthesis

- Work-in-progress
- Atom and Gates with VHDL *abstract syntax*

Simulation

- Combinational
- Sequential

Universiteit Utrecht

Examples

- `AndN`

Universiteit Utrecht

Problems

- Definition of $[\![\_]\!]$ blocks reduction

Summary

- Π-Ware is...

Universiteit Utrecht

Current limitations

- Problem with proofs (definition of $[\![\_]\!]$)
- Proofs on (infinite) Streams
- Bla

Future work

- Proof by reflection for finite cases

Universiteit Utrecht

Thank you!

Questions?

References I

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

Sheeran, M. (1984).
MuFP, a language for VLSI design.
In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 104–112, New York, NY, USA. ACM.

Wadler, P. (2014).
Propositions as types.
Unpublished note, `http://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf`.