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

Sunday 24th August, 2014

Universiteit Utrecht

Table of Contents

Background
Hardware Design
Functional Hardware
DTP
Research Question
Question
Method
DTP / Agda
Big picture
Agda
Π-Ware
Syntax
Semantics
Proofs
Conclusions
Limitations
Future work

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

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?"

Universiteit Utrecht

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
  - take : $(m : \mathbb{N}) \to \mathsf{Vec}\ \alpha\ (m + n) \to \mathsf{Vec}\ \alpha\ m$

Universiteit Utrecht

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:

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

    - This does not:

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

Universiteit Utrecht

Dependently-Typed Programming

- Dependent pattern matching can *rule out* impossible cases

Universiteit Utrecht

Dependently-Typed Programming

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

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

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 _::_

Depedent types as logic

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

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n}               → zero   ≤ n
  s≤s : ∀ {m n} → m ≤ n → suc m ≤ suc n
```

  - Proposition:

```
twoLEQFour : 2 ≤ 4
```

  - Proof:

```
twoLEQFour = s≤s (s≤s z≤n)
```

s≤s (s≤s (z≤n : 0 ≤ 4) : 1 ≤ 4) : 2 ≤ 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: ℂ, ⟦ c ⟧, ⇓, ⇑
- *Mixfix* notation
  - _[_]≔_ is the vector update function: v [# 3] ≔ true.
  - _[_]≔_ v (# 3) true ⟺ v [# 3] ≔ true
- Almost nothing built-in
  - _+_ : ℕ → ℕ → ℕ defined in Data.Nat
  - if_then_else_ : Bool → α → α → α defined in Data.Bool

Universiteit Utrecht

Agda syntax for Haskell programmers

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

Low-level circuits

- Structural representation
- Untyped but *sized*

$$
\begin{array}{l}
\mathsf{data}\ \mathbb{C}' : \mathbb{N} \to \mathbb{N} \to \mathsf{Set} \\
\mathsf{data}\ \mathbb{C}'\ \mathsf{where} \\
\quad \mathsf{Nil} \quad : \mathbb{C}'\ \mathsf{zero}\ \mathsf{zero} \\
\quad \mathsf{Gate} \quad : (g\# : \mathsf{Gates\#}) \to \mathbb{C}'\ (|\mathsf{in}|\ g\#)\ (|\mathsf{out}|\ g\#) \\
\quad \mathsf{Plug} \quad : \forall\ \{i\ o\} \to (f : \mathsf{Fin}\ o \to \mathsf{Fin}\ i) \to \mathbb{C}'\ i\ o \\
\\
\quad \mathsf{DelayLoop} : (c : \mathbb{C}'\ (i + l)\ (o + l))\ \{\mathsf{comb}'\ c\} \to \mathbb{C}'\ i\ o \\
\\
\quad \_\rangle\!\rangle'\_ \quad : \mathbb{C}'\ i\ m \to \mathbb{C}'\ m\ o \to \mathbb{C}'\ i\ o \\
\quad \_|'\_ \quad : \mathbb{C}'\ i_1\ o_1 \to \mathbb{C}'\ i_2\ o_2 \to \mathbb{C}'\ (i_1 + i_2)\ (o_1 + o_2) \\
\quad \_|+'\_ \quad : \mathbb{C}'\ i_1\ o \to \mathbb{C}'\ i_2\ o \to \mathbb{C}'\ (\mathsf{suc}\ (i_1 \sqcup i_2))\ o
\end{array}
$$

Atoms

- How to carry values of an Agda type in *one* wire
- Defined by the Atomic type class in PiWare.Atom

```
record Atomic : Set₁ where
  field
    Atom       : Set
    |Atom|-1   : ℕ
    n→atom     : Fin (suc |Atom|-1) → Atom
    atom→n     : Atom → Fin (suc |Atom|-1)

    inv-left   : ∀ i → atom→n (n→atom i) ≡ i
    inv-right  : ∀ a → n→atom (atom→n a) ≡ a

  |Atom| = suc |Atom|-1
  Atom#  = Fin |Atom|
```

Atomic instances

- Examples of types that can be Atomic
  - Bool, std_logic, other multi-valued logics
  - Predefined in the library: PiWare.Atom.Bool
- First, define how many atoms we are interested in

$$|\mathsf{B}|{-}1 = 1$$
$$|\mathsf{B}| = \mathsf{suc}\ |\mathsf{B}|{-}1$$

- Friendlier names for the indices (elements of Fin 2)

```
pattern False#   = Fz
pattern True#    = Fs Fz
pattern Absurd# n = Fs (Fs n)
```

Atomic instance (Bool)

- Bijection between $\{n \in \mathbb{N} \mid n < 2\}$ (Fin 2) and Bool

```
n→B = λ { False# → false;  True#  → true;  (Absurd# ()) }
B→n = λ { false  → False#; true   → True# }
```

- Proof that n→B and B→n are inverses

```
inv-left-B  = λ { False# → refl; True# → refl; (Absurd# ()) }
inv-right-B = λ { false  → refl; true  → refl }
```

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

Examples

- `AndN`

Problems

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

Summary

- Π-Ware is...

Current limitations

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

Universiteit Utrecht

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`.