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

Universiteit Utrecht

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

Universiteit Utrecht

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

Universiteit Utrecht

Functional Programming

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

Universiteit Utrecht

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]

Universiteit Utrecht

Embedded DSLs for Hardware

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

Universiteit Utrecht

Dependently-Typed Programming

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

Universiteit Utrecht

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

Universiteit Utrecht

Dependently-Typed Programming

▶ Types can depend on values

  - Example: data Vec (α : Set) : $\mathbb{N} \to$ 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 : $\mathbb{N}$) $\to$ Vec α ($m + n$) $\to$ Vec α 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:
      $$add : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$
      $$add \; zero \quad y \; = y$$
      $$add \; (suc \; x') \; y \; = suc \; (add \; x' \; y)$$
    - This does not:
      $$silly : \mathbb{N} \rightarrow \mathbb{N}$$
      $$silly \; zero \quad = zero$$
      $$silly \; (suc \; n') \; = silly \; \lfloor \; n' \; /2 \rfloor$$

Universiteit Utrecht

Dependently-Typed Programming

- Dependent pattern matching can *rule out* impossible cases

Universiteit Utrecht

Dependently-Typed Programming

Background
Hardware Design
Functional Hardware
DTP

Research
Question
Question
Method

DTP / Agda
Big picture
Agda

Π-Ware
Syntax
Semantics
Proofs

Conclusions
Limitations
Future work

▶ Dependent pattern matching can *rule out* impossible cases

  · Classic example: *safe* head function
    head : Vec α (suc n) $\rightarrow$ α
    head ($x :: xs$) = x

Universiteit Utrecht

Dependently-Typed Programming

▶ Dependent pattern matching can *rule out* impossible cases

- Classic example: *safe* head function
  head : Vec α (suc n) $\rightarrow$ α
  head (x :: xs) = x

- The **only** constructor returning Vec α (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):
    data $\_\leq\_$: $\mathbb{N} \to \mathbb{N} \to$ Set where
    $z{\leq}n$: $\forall \{n\}$ $\to$ zero $\leq n$
    $s{\leq}s$: $\forall \{m\ n\}$ $\to m \leq n \to$ suc $m \leq$ suc n
  - Proposition:
    twoLEQFour : $2 \leq 4$
  - Proof:
    twoLEQFour = $s{\leq}s$ ($s{\leq}s$ $z{\leq}n$)
    $s{\leq}s$ ($s{\leq}s$ ($z{\leq}n$: $0 \leq 4$) : $1 \leq 4$) : $2 \leq 4$

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: $\mathbb{C}$, $[\![\, c\,]\!]$, ⇓, ⇑

▶ *Mixfix* notation

  · _[_]:=_ is the vector update function: v [# 3] := true.
  · _[_]:=_ v (# 3) true ⟺ 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

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:

        · $\forall\ n \to \mathsf{zero} \leq n$

        · data $\_\leq\_ : \mathbb{N} \to \mathbb{N} \to \mathsf{Set}$

▶ It's common to combine both:

    · $\forall\ \{\alpha\ n\} \to \mathsf{Vec}\ \alpha\ (\mathsf{suc}\ n) \to \alpha \iff$
      $\{\alpha : \_\}\ \{n : \_\} \to \mathsf{Vec}\ \alpha\ n \to \alpha$

Universiteit Utrecht

Low-level circuits

- Structural representation
- Untyped but *sized*

Background
Hardware Design
Functional Hardware
DTP

Research
Question
Question
Method

DTP / Agda
Big picture
Agda

Π-Ware
Syntax
Semantics
Proofs

Conclusions
Limitations
Future work

```
data ℂ′ : ℕ → ℕ → Set
data ℂ′ where
   Nil    : ℂ′ zero zero
   Gate   : (g# : Gates#) → ℂ′ (|in| g#) (|out| g#)
   Plug   : ∀ {i o}          → (f : Fin o → Fin i) → ℂ′ i o

   DelayLoop : (c : ℂ′ (i + l) (o + l)) {comb′ c} → ℂ′ i o

   _⟫′_   : ℂ′ i m   → ℂ′ m o   → ℂ′ i o
   _|′_   : ℂ′ i₁ o₁ → ℂ′ i₂ o₂ → ℂ′ (i₁ + i₂) (o₁ + o₂)
   _|+′_  : ℂ′ i₁ o  → ℂ′ i₂ o  → ℂ′ (suc (i₁ ⊔ i₂)) o
```

Universiteit Utrecht

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|
```

Universiteit Utrecht

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

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

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

  pattern False# = Fz
  pattern True#   = Fs Fz

Universiteit Utrecht

Atomic instance (Bool)

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

$$n{\rightarrow}B = \lambda \; \{ \; \mathsf{False\#} \; \rightarrow \; false; \quad \mathsf{True\#} \; \rightarrow \; true \; \}$$
$$B{\rightarrow}n = \lambda \; \{ \; false \; \rightarrow \; \mathsf{False\#}; \quad true \quad \rightarrow \; \mathsf{True\#} \; \}$$

▶ Proof that $n{\rightarrow}B$ and $B{\rightarrow}n$ are inverses

$$\mathsf{inv{-}left{-}B} \;\; = \lambda \; \{ \; \mathsf{False\#} \; \rightarrow \; refl; \quad \mathsf{True\#} \; \rightarrow \; refl; \; \}$$
$$\mathsf{inv{-}right{-}B} \; = \lambda \; \{ \; false \quad \rightarrow \; refl; \quad true \quad \rightarrow \; refl \; \}$$

▶ With all pieces at hand, we construct the instance

$$
\begin{aligned}
\mathsf{Atomic{-}B} = \mathsf{record} \; \{ \; &\mathsf{Atom} &&= \mathsf{B} \\
; \; &|\mathsf{Atom}|{-}1 &&= |\mathsf{B}|{-}1 \\
; \; &\mathsf{n{\rightarrow}atom} &&= \mathsf{n{\rightarrow}B} \\
; \; &\mathsf{atom{\rightarrow}n} &&= \mathsf{B{\rightarrow}n} \\
; \; &\mathsf{inv{-}left} &&= \mathsf{inv{-}left{-}B} \\
; \; &\mathsf{inv{-}right} &&= \mathsf{inv{-}right{-}B} \; \}
\end{aligned}
$$

Universiteit Utrecht

Gates

▶ Circuits parameterized by collection of *fundamental gates*
▶ Examples:
  · {NOT, AND, OR} (BoolTrio)
  · {NAND}
  · Arithmetic, Crypto, etc.

▶ The definition of what means to be such a collection is in PiWare.Gates.Gates

Universiteit Utrecht

The Gates type class

```
W : ℕ → Set
W = Vec Atom
 record Gates : Set where

    field
       |Gates|    : ℕ
       |in| |out| : Fin |Gates| → ℕ
       spec       : (g : Fin |Gates|)
                    → (W (|in| g) → W (|out| g))

    Gates#  = Fin |Gates|
```

Universiteit Utrecht

Gates instances

- Example: PiWare.Gates.BoolTrio
- First, how many gates are there in the library

  $|\text{BoolTrio}| = 5$

- Then the friendlier names for the indices

  pattern FalseConst# = Fz
  pattern TrueConst#   = Fs Fz
  pattern Not#   = Fs (Fs Fz)
  pattern And#   = Fs (Fs (Fs Fz))
  pattern Or#   = Fs (Fs (Fs (Fs Fz)))

Universiteit Utrecht

Gates instance (BoolTrio)

▶ Defining the *interfaces* of the gates

|in| FalseConst# = 0
|in| TrueConst#   = 0
|in| Not#   = 1
|in| And#   = 2
|in| Or#   = 2

|out| _ = 1

▶ And the specification function for each gate

$$
\begin{aligned}
\text{spec−false} \quad \_ &= [\ false\ ] \\
\text{spec−true} \quad \_ &= [\ true\ ] \\
\text{spec−not} \quad (x :: \varepsilon) &= [\ not\ x\ ] \\
\text{spec−and} \quad (x :: y :: \varepsilon) &= [\ x \wedge y\ ] \\
\text{spec−or} \quad (x :: y :: \varepsilon) &= [\ x \vee y\ ]
\end{aligned}
$$

Universiteit Utrecht

Gates instance (BoolTrio)

▶ Mapping each gate index to its respective specification

```
specs−BoolTrio FalseConst#  = spec−false
specs−BoolTrio TrueConst#   = spec−true
specs−BoolTrio Not#         = spec−not
specs−BoolTrio And#         = spec−and
specs−BoolTrio Or#          = spec−or
```

▶ With all pieces at hand, we construct the instance

```
BoolTrio : Gates
BoolTrio = record  { |Gates| = |BoolTrio|
                   ; |in|    = |in|
                   ; |out|   = |out|
                   ; spec    = specs−BoolTrio }
```

Universiteit Utrecht

High-level circuits

- User is not supposed to describe circuits at low level ($\mathbb{C}'$)
- The high level circuit type ($\mathbb{C}$) alloes for *typed* circuit interfaces
  - The input and output indices are Agda types

$$\text{data } \mathbb{C}\ (\alpha\ \beta\ :\ \mathsf{Set})\ \{i\ j\ :\ \mathbb{N}\}\ :\ \mathsf{Set}\ \text{where}$$
$$\mathsf{Mk}\mathbb{C}\ :\ \{\!\!|\ s\alpha\ :\ \Downarrow\!\mathsf{W}\!\!\Uparrow\ \alpha\ \{i\}\ |\!\!\}\ \{\!\!|\ s\beta\ :\ \Downarrow\!\mathsf{W}\!\!\Uparrow\ \beta\ \{j\}\ |\!\!\}$$
$$\to\ \mathbb{C}'\ i\ j\ \to\ \mathbb{C}\ \alpha\ \beta\ \{i\}\ \{j\}$$

- $\mathsf{Mk}\mathbb{C}$ takes:
  - Low level description ($\mathbb{C}'$)
  - Information on how to *synthesize* elements of α and β
    - Passed as *instance arguments*

Universiteit Utrecht

Synthesizable

▶ ⇓W⇑ type class (pronounced `Synthesizable`)

- Describes how to *synthesize* a given Agda type (α)
- Two fields: from element of α to a *word* and back

```
record ⇓W⇑ (α : Set) {i : ℕ} : Set where
  constructor ⇓W⇑[_,_]
  field
    ⇓ : α → W i
    ⇑ : W i → α
```

Universiteit Utrecht

⇓W⇑ instances

- Any *finite* type can have such an instance
- Predefined in the library: Bool; _×_; _⊎_; Vec
- Example: instance for products (_×_)

$$⇓W⇑−× \; : \; \{\!\!\{ \; s\alpha \; : \; ⇓W⇑ \; \alpha \; \{i\} \; \}\!\!\} \; \{\!\!\{ \; s\beta \; : \; ⇓W⇑ \; \beta \; \{j\} \; \}\!\!\}$$
$$\qquad \qquad → ⇓W⇑ \; (\alpha × \beta)$$

$$⇓W⇑−× \; \{\alpha\} \; \{i\} \; \{\beta\} \; \{j\} \; \{\!\!\{ \; s\alpha \; \}\!\!\} \; \{\!\!\{ \; s\beta \; \}\!\!\} = ⇓W⇑[\; \text{down} \; , \; \text{up} \;]$$
$$\quad \text{where} \; \; \text{down} \; : \; (\alpha × \beta) → W \; (i + j)$$
$$\qquad \qquad \text{down} \; (a \; , \; b) = (⇓ \; a) ++ (⇓ \; b)$$

$$\qquad \qquad \text{up} \; : \; W \; (i + j) → (\alpha × \beta)$$
$$\qquad \qquad \text{up} \; w \; \text{with} \; \text{splitAt} \; i \; w$$
$$\qquad \qquad \text{up} \; .(⇓a ++ ⇓b) \; | \; ⇓a \; , \; ⇓b \; , \; \text{refl} = ⇑ \; ⇓a \; , \; ⇑ \; ⇓b$$

Universiteit Utrecht

Synthesizable

▶ Both fields ⇓ and ⇑ should be inverses of each other

Universiteit Utrecht

Circuit semantics

- ▶ *Synthesis* semantics: produce a *netlist*
  - · Tool integration / implement in FPGA or ASIC.

- ▶ *Simulation* semantics: *execute* a circuit
  - · Given circuit model and inputs, calculate outputs

- ▶ Other semantics possible:
  - · Timing analysis, power estimation, etc.
  - · This possibility guided Π-Ware's development
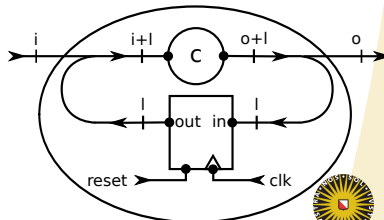
Universiteit Utrecht

Synthesis semantics

▶ Netlist: digraph with *gates* as nodes and *buses* as edges

$$\text{Nil} : \mathbb{C} \; 0 \; 0$$

$$\frac{\text{i o} : \mathbb{N} \quad \text{f} : \text{Fin o} \to \text{Fin i}}{\text{Plug f} : \mathbb{C} \; \text{i} \; \text{o}}$$

$$\frac{\text{g\#} : \text{Gate\#}}{\text{Gate g\#} : \mathbb{C} \; (\text{ins g\#}) \; (\text{outs g\#})}$$

$$\frac{\text{c} : \mathbb{C} \; (\text{i+l}) \; (\text{o+l})}{\text{DelayLoop} : \mathbb{C} \; \text{i} \; \text{o}}$$

Background
Hardware Design
Functional Hardware
DTP

Research
Question
Question
Method

DTP / Agda
Big picture
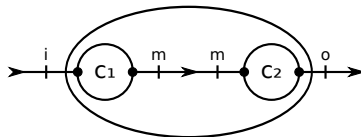Agda

Π-Ware
Syntax
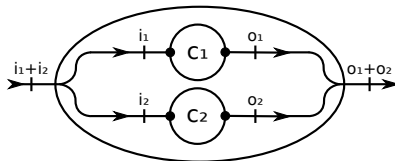**Semantics**
Proofs

Conclusions
Limitations
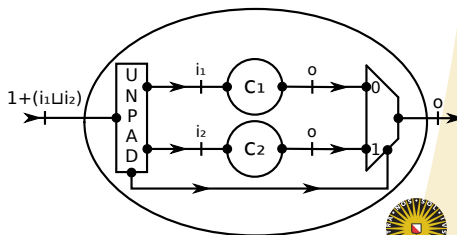Future work

Universiteit Utrecht

Synthesis semantics

$$c_1 : \mathbb{C} \; i \; m$$
$$c_2 : \mathbb{C} \; m \; o$$
$$\overline{c_1 \rangle\!\rangle' \; c_2 : \mathbb{C} \; i \; o}$$

$$c_1 : \mathbb{C} \; i_1 \; o_1$$
$$c_2 : \mathbb{C} \; i_2 \; o_2$$
$$\overline{c_1 \;|' \; c_2 : \mathbb{C} \; (i_1+i_2) \; (o_1+o_2)}$$

$$c_1 : \mathbb{C} \; i_1 \; o$$
$$c_2 : \mathbb{C} \; i_2 \; o$$
$$\overline{c_1 \;|+' \; c_2 : \mathbb{C} \; (1+(i_1 \sqcup i_2)) \; o}$$

Universiteit Utrecht

Synthesis semantics

Missing "pieces":

- ▶ Adapt Atomic

  - New field: a VHDLTypeDecl

    - Such as: **type** ident **is** (elem1, elem2);
    - Enumerations, integers (ranges), records.

  - New field: atomVHDL : Atom → VHDLExpr

- ▶ Adapt Gates

  - For each gate, a corresponding VHDLEntity
  - netlist : ($g\#$: Gates) → VHDLEntity (|in| $g\#$) (|out| $g\#$)

    - The VHDL entity has the *interface* of corresponding gate

Universiteit Utrecht

Simulation semantics

- ▶ Two levels of abstraction
  - · High-level simulation ($[\![\_]\!]$) for high-level circuits ($\mathbb{C}$)
  - · Low-level simulation ($[\![\_]\!]'$) for low-level circuits ($\mathbb{C}'$)

- ▶ Two kinds of simulation
  - · Combinational simulation ($[\![\_]\!]$) for stateless circuits
  - · Sequential simulation ($[\![\_]\!]*$) for stateful circuits

- ▶ High level defined in terms of low level

$$[\![\_]\!] : \forall \{\alpha\ i\ \beta\ j\} \rightarrow (c : \mathbb{C}\ \alpha\ \beta\ \{i\}\ \{j\}) \rightarrow (\alpha \rightarrow \beta)$$
$$[\![\ \mathsf{Mk}\mathbb{C}\ \{\!|\ s\alpha\ |\!\}\ \{\!|\ s\beta\ |\!\}\ c'\ ]\!] = \Uparrow \circ [\![\ c'\ ]\!]' \circ \Downarrow$$

Universiteit Utrecht

Examples

▶ AndN

Universiteit Utrecht

Problems

▶ Definition of ⟦_⟧ blocks reduction

Universiteit Utrecht

Summary

▶ Π-Ware is...

Universiteit Utrecht

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`.

Universiteit Utrecht