Π-Ware: Hardware Description with Dependent Types

João Paulo Pizani Flor, B.Sc
<j.p.pizaniflor@students.uu.nl>

dr. W. S. Swierstra
<w.s.swierstra@uu.nl>

Department of Information and Computing Sciences
Utrecht University

Saturday 12th July, 2014

Universiteit Utrecht

Table of Contents

Section 1

Intro

Universiteit Utrecht

One-sentence definition

Π-Ware is a Domain-Specific Language (DSL) embedded in Agda for *modeling* hardware, *synthesizing* it and *reasoning* about its properties.

Universiteit Utrecht

Hardware Design

Hardware design is a *complex* and *"booming"* activity:

- Algorithms increasingly benefit from *hardware acceleration*
  - Moore's Law still holds
  - Microarchitecture optimization has diminishing returns
- Hardware development has stricter requirements
  - Mistakes found in "production" are much more serious
  - Thus the need for extensive validation/verification
    - Can encompass up to 50% of total development costs
- Need to combine productivity/ease-of-use with rigor
  - Detect mistakes *early*

Hardware design

Functional programming has already been used to help hardware design (since the 1980s).

- First, *independent* DSLs (e.g. muFP)
- Then, as *embedded* DSLs
  - Prominently, in Haskell
- Question: How to use DTP to benefit hardware design?
  - Experimenting by embedding in a DTP language
  - Namely, Π-Ware is embedded in *Agda* (ITT, Martin-Löf)

Some features of Agda important for us

Not *exclusively*...

- Dependent inductive families
  - Circuits are indexed by the sizes/types of their ports
- Dependent pattern matching
- "Dependent type classes"
  - Dependent records + instance arguments
- Coinductive types / proofs
  - When modeling / proving sequential behaviour
- Parameterized modules

Credit where credit is due

- Lava – *Haskell* (Chalmers)
  - Pragmatic, easy-to-use, popular
- ForSyDe – *Haskell* (KTH)
  - Hierarchical synthesis
  - Static size checking
- Coquet – *Coq* (INRIA)
  - *Main* influence
    - Reasoning about circuit behaviour with Coq's tactics
    - Models circuits with structural combinators

Universiteit Utrecht

Section 2

Dive into Π-Ware

Universiteit Utrecht

Modeling circuits

- Deep-embedded – explicit circuit inductive family: $\mathbb{C}'$
- Descriptions are at *gate level* and *architectural*
  - Fundamental constructors: Nil, Gate (parameterized)
  - Constructors for structural combination

```
data ℂ′ : ℕ → ℕ → Set

data ℂ′ where

  Nil  : ℂ′ zero zero

  Gate : (g# : Gates#) → ℂ′ (ins g#) (outs g#)

  Plug : {i o : ℕ} → (f : Fin o → Fin i) → ℂ′ i o

  _⟫′_ : {i m o : ℕ} → ℂ′ i m → ℂ′ m o → ℂ′ i o

  _|′_ : {i₁ o₁ i₂ o₂ : ℕ} → ℂ′ i₁ o₁ → ℂ′ i₂ o₂ → ℂ′ (i₁ + i₂) (o₁ + o₂)

  _|+′_ : {i₁ i₂ o : ℕ} → ℂ′ i₁ o → ℂ′ i₂ o → ℂ′ (suc (i₁ ⊔ i₂)) o
```


Universiteit Utrecht

Sequential circuits

- Built using DelayLoop, introduces state (latch)

$$\begin{aligned}&\mathsf{DelayLoop} : \{i\ o\ l : \mathbb{N}\}\ (c : \mathbb{C}'\ (i + l)\ (o + l))\\&\quad \{p : \mathsf{comb}'\ c\} \to \mathbb{C}'\ i\ o\end{aligned}$$

- Avoid evaluating combinational loops by carrying a proof

$$\begin{aligned}&\mathsf{comb}' : \{i\ o : \mathbb{N}\} \to \mathbb{C}'\ i\ o \to \mathsf{Set}\\&\mathsf{comb}'\ \mathsf{Nil} = \top\\&\mathsf{comb}'\ (\mathsf{Gate}\ \_) = \top\\&\mathsf{comb}'\ (\mathsf{Plug}\ \_) = \top\\&\mathsf{comb}'\ (\mathsf{DelayLoop}\ \_) = \bot\\&\mathsf{comb}'\ (c_1 \ggg' c_2) = \mathsf{comb}'\ c_1 \times \mathsf{comb}'\ c_2\end{aligned}$$

Universiteit Utrecht

Abstraction (Atomic)

- Circuits operate over *words*, which are made of Atom
  - PiWare "ships" with Bool atoms
  - Another example: IEEE1164 multi-valued (`std_logic`)

```
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
```


Computation abstraction (Gates)

- Circuits parameterized by a record of "fundamental" Gates
  - Specifying each gate's *interface* (input/output sizes)
  - And its *semantics* (as a function over *words*)

```
record Gates : Set where
  field
    |Gates| – 1 : ℕ
    ins outs    : Fin (suc |Gates| – 1) → ℕ
    spec        : (g : Fin (suc |Gates| – 1)) → (W (ins g) → W (outs g))

  |Gates| = suc |Gates| – 1
  Gates#  = Fin |Gates|
```

- "Black box" for semantics / reasoning
  - Correctness assumed

Universiteit Utrecht

Data abstraction (input/output)

- The "core" circuit type ($\mathbb{C}'$) is indexed by sizes ($\mathbb{N}$)
  - Has *words* of the respective sizes as inputs and outputs
- The high level type ($\mathbb{C}$) is indexed by *meta* (Agda) types
  - Specifically, *finite* types

```
data ℂ (α β : Set) {i j : ℕ} : Set where
  Mkℂ : ⦃ sα : ⇓ W ⇑ α {i} ⦄ ⦃ sβ : ⇓ W ⇑ β {j} ⦄
      → ℂ′ i j → ℂ α β {i} {j}

comb : ∀ {α i β j} → ℂ α β {i} {j} → Set
comb (Mkℂ c′) = comb′ c′
```


Universiteit Utrecht

Data abstraction (Synthesizable)

- We define a *class* of *finite types*
  - Practically, they are types which can be mapped to *words*
  - The isomorphism resides in the Synthesizable type class

```
W : ℕ → Set
W = Vec Atom

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

- Instances for _×_, _⊎_, Vec, Bool
  - Lack recursive instance search

Simulation semantics

- "Purely combinational" vs. sequential
- *Simulation* functions in 2 levels of abstraction
  - Low-level eval: $\mathbb{C}'\ i\ o \to (\mathsf{W}\ i \to \mathsf{W}\ o)$
  - High-level eval: $\mathbb{C}\ \alpha\ \beta \to (\alpha \to \beta)$

$$[\![\_]\!]' : \{i\ o : \mathbb{N}\} \to (c : \mathbb{C}'\ i\ o)\ \{p : \mathsf{comb}'\ c\} \to (\mathsf{W}\ i \to \mathsf{W}\ o)$$

$$[\![\ \mathsf{Nil}\ ]\!]' = \mathsf{const}\ \varepsilon$$

$$[\![\ \mathsf{Gate}\ g\#\ ]\!]' = \mathsf{spec}\ g\#$$

$$[\![\ \mathsf{Plug}\ p\ ]\!]' = \mathsf{plugOutputs}\ p$$

$$[\![\ c_1 \ggg' c_2\ ]\!]'\ \{p_1 , p_2\} = [\![\ c_2\ ]\!]'\ \{p_2\} \circ [\![\ c_1\ ]\!]'\ \{p_1\}$$

$$[\![\ \_|'\_\ \{i_1\}\ c_1\ c_2\ ]\!]'\ \{p_1 , p_2\} =$$
$$\mathsf{uncurry}'\ \_++\_ \circ \mathsf{map}\ ([\![\ c_1\ ]\!]'\ \{p_1\})\ ([\![\ c_2\ ]\!]'\ \{p_2\}) \circ \mathsf{splitAt}'\ i_1$$

$$[\![\ \_|+'\_\ \{i_1\}\ c_1\ c_2\ ]\!]'\ \{p_1 , p_2\} = [\ [\![\ c_1\ ]\!]'\ \{p_1\} , [\![\ c_2\ ]\!]'\ \{p_2\}\ ]' \circ \mathsf{untag}\ \{i_1\}$$

$$[\![\ \mathsf{DelayLoop}\ c\ ]\!]'\ \{()\}\ v$$

Simulation semantics

- "High-level" simulation has a pretty simple definition

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

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

- Let's go over sequential simulation in a bit more detail...

Universiteit Utrecht

Sequential simulation

- Modeled using *causal stream functions* (Uustalu, 2006)
  - Output depends on current and previous inputs, *not* future
  - Implemented in Agda as non-empty lists (List^+)

$$
\begin{aligned}
&[\![\_]\!] : \{i\ o : \mathbb{N}\} \to \mathbb{C}'\ i\ o \to (\mathsf{W}\ i \Rightarrow \mathsf{W}\ o)\\
&[\![\ \mathsf{Nil}\ ]\!]\ (w^0\ ,\ \_) = [\![\ \mathsf{Nil}\ ]\!]'\ w^0\\
&[\![\ \mathsf{Gate}\ g\#\ ]\!]\ (w^0\ ,\ \_) = [\![\ \mathsf{Gate}\ g\#\ ]\!]'\ w^0\\
&[\![\ \mathsf{Plug}\ p\ ]\!]\ (w^0\ ,\ \_) = \mathsf{plugOutputs}\ p\ w^0\\
&[\![\ \mathsf{DelayLoop}\ \{\mathsf{o} = j\}\ c\ ]\!] = \mathsf{take}_{\mathsf{v}}\ j \circ \mathsf{delay}\ \{\mathsf{o} = j\}\ c\\
&[\![\ c_1 \ggg' c_2\ ]\!] = [\![\ c_2\ ]\!] \circ \mathsf{map}^+\ [\![\ c_1\ ]\!] \circ \mathsf{tails}^+\\
&[\![\ \_|'\_\ \{i_1\}\ c_1\ c_2\ ]\!] = \mathsf{uncurry}'\ \_{+}{+}\_ \circ \mathsf{map}\ [\![\ c_1\ ]\!]\ [\![\ c_2\ ]\!] \circ \mathsf{unzip}^+ \circ \mathsf{splitAt}^+\ i_1\\
&[\![\ \_|+'\_\ \{i_1\}\ c_1\ c_2\ ]\!]\ (w^0\ ,\ w^-) \text{ with } \mathsf{untag}\ \{i_1\}\ w^0 \mid \mathsf{untagList}\ \{i_1\}\ w^-\\
&\ldots \mid \mathsf{inj}_1\ w_1^0 \mid w_1^-\ ,\ \_ = [\![\ c_1\ ]\!]\ (w_1^0\ ,\ w_1^-)\\
&\ldots \mid \mathsf{inj}_2\ w_2^0 \mid \_\ ,\ w_2^- = [\![\ c_2\ ]\!]\ (w_2^0\ ,\ w_2^-)
\end{aligned}
$$

Universiteit Utrecht

Sequential simulation

- We "run" the causal eval to get a Stream based eval

$$\Gamma : (\alpha : \mathsf{Set}) \to \mathsf{Set}$$
$$\Gamma = \mathsf{List}^{+}$$

$$\_ \Rightarrow \_ : (\alpha\ \beta : \mathsf{Set}) \to \mathsf{Set}$$
$$\alpha \Rightarrow \beta = \Gamma\ \alpha \to \beta$$

$$\mathsf{run} : \forall\ \{\alpha\ \beta\} \to (\alpha \Rightarrow \beta) \to (\mathsf{Stream}\ \alpha \to \mathsf{Stream}\ \beta)$$
$$\mathsf{run}\ f\ (x^0 :: x^+) = \mathsf{run}'\ f\ ((x^0\ ,\ [])\ ,\ \flat\ x^+) \ \mathsf{where}$$
$$\mathsf{run}' : \forall\ \{\alpha\ \beta\} \to (\alpha \Rightarrow \beta) \to (\Gamma\ \alpha \times \mathsf{Stream}\ \alpha) \to \mathsf{Stream}\ \beta$$
$$\mathsf{run}'\ f\ ((x^0\ ,\ x^-)\ ,\ (x^1 :: x^+)) =$$
$$f\ (x^0\ ,\ x^-) :: \sharp\ \mathsf{run}'\ f\ ((x^1\ ,\ x^0 :: x^-)\ ,\ \flat\ x^+)$$

$$[\![\_]\!]*' : \{i\ o : \mathbb{N}\} \to \mathbb{C}'\ i\ o \to (\mathsf{Stream}\ (\mathsf{W}\ i) \to \mathsf{Stream}\ (\mathsf{W}\ o))$$
$$[\![\_]\!]*' = \mathsf{run} \circ [\![\_]\!]$$

Universiteit Utrecht

Reasoning about circuit properties

- Use ⟦_⟧ and ⟦_⟧* to express circuit *behaviour*
- Functional correctness: equality with some *specification*
  - Also depends on the Synthesizable instances
  - Could benefit from proof automation (case analysis, etc.)
  - Investigating "proof combinators"

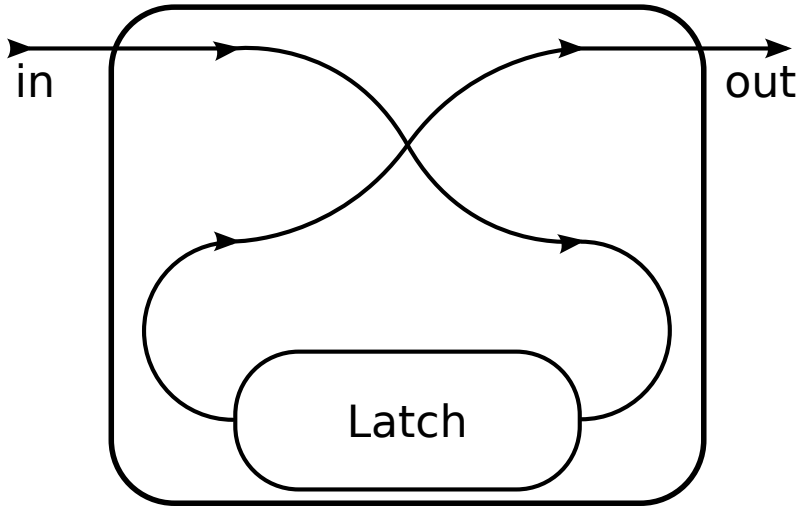
```
proofFaddBool : ∀ a b c → ⟦ fadd ⟧ ((a , b) , c) ≡ faddSpec a b c
proofFaddBool true  true  true  = refl
proofFaddBool true  true  false = refl
proofFaddBool true  false true  = refl
proofFaddBool true  false false = refl

proofHaddBool : ∀ a b → ⟦ hadd ⟧ (a , b) ≡ haddSpec a b
proofHaddBool a b = cong (_, _ (a ∧ b)) (xorEquiv a b)
```


Universiteit Utrecht

Properties of sequential circuits

- Defining correctness of sequential circuits is not so trivial
- One (very simple) example: a shift register

```
shift : ℂ B B
shift = delayℂ pSwap
```



- Currently working in this area

Universiteit Utrecht

Compiling to VHDL

- Π-Ware shall support compiling circuits into VHDL netlists
  - Main goal: generate *synthesizable* (IEEE 1076.3)
  - Secondary: hierarchical descriptions (components)
- Work in progress
  - More critical problems to be solved first

Compiling to VHDL

Requirements for VHDL generation:

- The DSL must be *deep-embedded*
- "Fundamental" gates need to have a structural definition
  - Extra field of Gates
  - Mapping each gate to a piece of VHDL abstract syntax.
- To support hierarchical modeling:
  - Some form of "component" declaration in the circuit types
    - Investigate approaches for naming
    - Reflection could help

Section 3

Current / next steps

Universiteit Utrecht

Current work

- Proof properties of sequential circuits
  - Using *bisimilarity*
  - For example, for the shift register we have seen:

```
shift : ℂ B B
shift = delayℂ pSwap

proofShiftTail : ∀ {ins} → tail ([[ shift ]]* ins) ≈ ins
proofShiftTail = undefined
```


Current work

- Proof combinators

$$
\begin{aligned}
&\_\rangle\!\rangle \equiv' \_ : \{i\ m\ o : \mathbb{N}\}\ \{f_1 : \mathsf{W}\ i \to \mathsf{W}\ m\}\ \{f_2 : \mathsf{W}\ m \to \mathsf{W}\ o\} \\
&\quad \{c_1 : \mathbb{C}'\ i\ m\}\ \{c_2 : \mathbb{C}'\ m\ o\}\ \{p_1 : \mathsf{comb}'\ c_1\}\ \{p_2 : \mathsf{comb}'\ c_2\} \\
&\quad \to (\forall\ v_1 \to [\![\_]\!]'\ \{i\}\ \{m\}\ c_1\ \{p_1\}\ v_1 \equiv f_1\ v_1) \\
&\quad \to (\forall\ v_2 \to [\![\_]\!]'\ \{m\}\ \{o\}\ c_2\ \{p_2\}\ v_2 \equiv f_2\ v_2) \\
&\quad \to (\forall\ v \to [\![\_]\!]'\ \{i\}\ \{o\}\ (c_1\ \rangle\!\rangle'\ c_2)\ \{p_1\ \mathsf{comb}\rangle\!\rangle'\ p_2\}\ v \equiv (f_2 \circ f_1)\ v) \\
&\_\rangle\!\rangle \equiv' \_\ \{\mathsf{f}_1 = f_1\}\ pc_1\ pc_2\ v\ \mathsf{rewrite}\ \mathsf{sym}\ (pc_2\ (f_1\ v))\ |\ \mathsf{sym}\ (pc_1\ v) = \mathsf{refl}
\end{aligned}
$$

Universiteit Utrecht

Next steps

- VHDL generation
- Proof automation
  - Case analysis, boring proofs – reflection
  - Better instance search
    - `Auto` (Kokke, Swierstra)

Universiteit Utrecht

Thank you!

Questions?

Universiteit Utrecht