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

Tuesday 26th August, 2014

Universiteit Utrecht

Table of Contents

Introduction
What is Π-Ware
Background

Research
Question
Research Question /
Methodology

DTP / Agda
Big picture
Agda

Π-Ware
Syntax
Semantics
Proofs

Conclusions
Limitations
Future work

Universiteit Utrecht

What is Π-Ware

▶ Π-Ware är en…

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
  - Simulation / Synthesis / Verification

- ▶ Limitations
  - Low level types
  - No *size* checks

Universiteit Utrecht

Dependently-Typed Programming

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

Universiteit Utrecht

Research Question / Methodology

▶ **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

Universiteit Utrecht

Dependently-Typed Programming

- **Disclaimer:** Suspend disbelief in syntax
    - Examples are in *Agda*
    - Syntax similar to Haskell, details further ahead
- 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 (heuristics)

- Structurally-decreasing recursion
- This passes the check:

  add : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$
  add zero $\quad y = y$
  add (suc x') $\ y =$ suc (add $x'\ y$)

- This does not:

  silly : $\mathbb{N} \to \mathbb{N}$
  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
  - Classic example: *safe* head function
    $$\text{head} \;:\; \text{Vec } \alpha \text{ (suc } n) \to \alpha$$
    $$\text{head } (x :: xs) = x$$
  - 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:
    ```
    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

Introduction
What is Π-Ware
Background

Research
Question
Research Question /
Methodology

DTP / Agda
Big picture
Agda

Π-Ware
Syntax
Semantics
Proofs

Conclusions
Limitations
Future work

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
    - Used a lot 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
    - _+_ : $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ defined in Data.Nat
    - if_then_else_ : Bool $\rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ 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: ε : $\{\alpha : \mathsf{Set}\} \to \mathsf{Vec}\ \alpha\ \mathsf{zero}$

▶ "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*

```
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
  - Need at least 1 (later why)

```
|B|−1 = 1
|B| = 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

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

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

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

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

  $inv{-}right{-}B \;\; = \lambda \; \{\; false \quad \rightarrow refl; \quad true \quad \rightarrow refl \;\}$

- With all pieces at hand, we construct the instance

  $Atomic{-}B = record \; \{\; Atom \qquad\quad = B$

  $\qquad\qquad\qquad\qquad ; \; |Atom|{-}1 \;\; = |B|{-}1$

  $\qquad\qquad\qquad\qquad ; \; n{\rightarrow}atom \quad = n{\rightarrow}B$

  $\qquad\qquad\qquad\qquad ; \; atom{\rightarrow}n \quad = B{\rightarrow}n$

  $\qquad\qquad\qquad\qquad ; \; inv{-}left \quad = inv{-}left{-}B$

  $\qquad\qquad\qquad\qquad ; \; inv{-}right \quad = inv{-}right{-}B \;\}$

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

$W : \mathbb{N} \to \mathsf{Set}$

$W = \mathsf{Vec\ Atom}$

record Gates : Set where

    field

       $|Gates|$: $\mathbb{N}$

       $|in|\ |out|$: $\mathsf{Fin}\ |Gates| \to \mathbb{N}$

       spec   : $(g : \mathsf{Fin}\ |Gates|)$

                 $\to (W\ (|in|\ g) \to W\ (|out|\ g))$

$\mathsf{Gates\#} = \mathsf{Fin}\ |Gates|$

Universiteit Utrecht

Gates instances

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

  $|BoolTrio| = 5$

- ▶ 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

$$|in| \ \text{FalseConst\#} = 0$$
$$|in| \ \text{TrueConst\#} = 0$$
$$|in| \ \text{Not\#} = 1$$
$$|in| \ \text{And\#} = 2$$
$$|in| \ \text{Or\#} = 2$$

$$|out| \ \_ = 1$$

▶ And the specification function for each gate

$$\text{spec\!-\!false} \quad \_ \qquad\qquad = [\ false \ ]$$
$$\text{spec\!-\!true} \quad \_ \qquad\qquad = [\ true \ ]$$
$$\text{spec\!-\!not} \quad (x :: \varepsilon) \qquad = [\ not \ x \ ]$$
$$\text{spec\!-\!and} \quad (x :: y :: \varepsilon) = [\ x \wedge y \ ]$$
$$\text{spec\!-\!or} \quad (x :: y :: \varepsilon) = [\ x \vee y \ ]$$

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 ($\mathbb{C}$) allows for *typed* circuit interfaces
  - Input and output indices are Agda types

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

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

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 (_×_)

$$⇓W⇑-× : \{\![\ sα : ⇓W⇑\ α\ \{i\}\ ]\!\}\ \{\![\ sβ : ⇓W⇑\ β\ \{j\}\ ]\!\}$$
$$→ ⇓W⇑\ (α × β)$$
$$⇓W⇑-×\ \{α\}\ \{i\}\ \{β\}\ \{j\}\ \{\![\ sα\ ]\!\}\ \{\![\ sβ\ ]\!\} = ⇓W⇑[\ \text{down}\ ,\ \text{up}\ ]$$
where down : $(α × β) → W\ (i + j)$
       down $(a\ ,\ b) = (⇓\ a)\ ++\ (⇓\ b)$

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

Universiteit Utrecht

Synthesizable

▶ Both fields $\Downarrow$ and $\Uparrow$ 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
- Synthesis semantics: given netlists of subcircuits, build combination

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

$$\frac{\begin{array}{c} i\ o : \mathbb{N} \\ f : \text{Fin}\ o \to \text{Fin}\ i \end{array}}{\text{Plug}\ f : \mathbb{C}\ i\ o}$$

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

$$\frac{c : \mathbb{C}\ (i+l)\ (o+l)}{\text{DelayLoop} : \mathbb{C}\ i\ o}$$
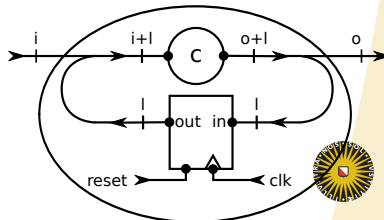
Introduction
What is Π-Ware
Background

Research
Question
Research Question /
Methodology

DTP / Agda
Big picture
Agda

Π-Ware
Syntax
Semantics
Proofs

Conclusions
Limitations
Future work

Universiteit Utrecht

Synthesis semantics

Introduction
What is Π-Ware
Background

Research
Question
Research Question /
Methodology

DTP / Agda
Big picture
Agda

Π-Ware
Syntax
Semantics
Proofs

Conclusions
Limitations
Future work

$$\frac{c_1 : \mathbb{C}\ i\ m \quad c_2 : \mathbb{C}\ m\ o}{c_1 \ggg' c_2 : \mathbb{C}\ i\ o}$$
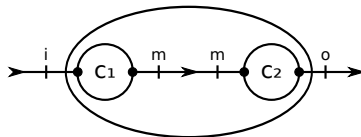
$$\frac{c_1 : \mathbb{C}\ i_1\ o_1 \quad c_2 : \mathbb{C}\ i_2\ o_2}{c_1\ |'\ c_2 : \mathbb{C}\ (i_1+i_2)\ (o_1+o_2)}$$

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

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

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

Universiteit Utrecht

Combinational simulation (excerpt)

$$[\![\_]\!]' : \forall \{i\ o\} \to (c : \mathbb{C}'\ i\ o)\ \{p : \mathsf{comb}'\ c\} \to (\mathsf{W}\ i \to \mathsf{W}\ o)$$
$$[\![\ \mathsf{Nil}\ ]\!]' = \mathsf{const}\ \varepsilon$$
$$[\![\ \mathsf{Gate}\ g\#\ ]\!]' = \mathsf{spec}\ g\#$$
$$[\![\ \mathsf{Plug}\ p\ ]\!]' = \mathsf{plugOutputs}\ p$$

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

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

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

▶ Remarks:

- Proof requires c to be combinational
- Gate case uses specification function
- DelayLoop case can be *discharged*

Universiteit Utrecht

Sequential simulation

Introduction
What is Π-Ware
Background

Research
Question
Research Question /
Methodology

DTP / Agda
Big picture
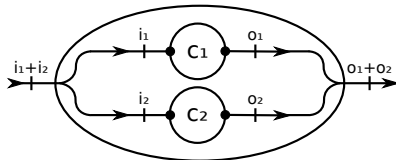Agda

Π-Ware
Syntax
Semantics
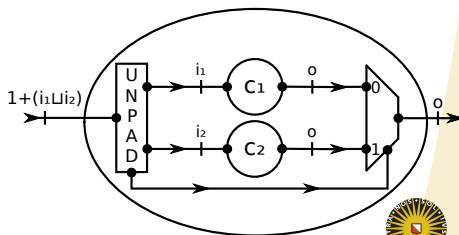Proofs

Conclusions
Limitations
Future work

▶ Inputs and outputs become Streams

- $\mathbb{C}' \ i \ o \implies$ Stream (W i) → Stream (W o)
- Stream: infinite list

▶ We can't write a recursive evaluation function over Streams

- *Sum* case (_|+'_) needs (Stream ($\alpha \uplus \beta$) → Stream α × Stream β)

  - What if there are no *lefts* (or *rights*)?

▶ A stream function is not an accurate model for hardware

- A function of type (Stream α → Stream β) can "look ahead"
- For example, tail ($x_0 :: x_1 :: x_2 :: xs$) = $x_1 :: x_2 :: xs$

Universiteit Utrecht

Causal stream functions

Solution: sequential simulation based on a *causal* stream function

Some definitions:

- ▶ Causal context: past + present values

  $\Gamma c : (\alpha : \text{Set}) \to \text{Set}$
  $\Gamma c \ \alpha = \alpha \times \text{List} \ \alpha$

- ▶ Causal stream function: produces **one** (current) output

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

Introduction
What is Π-Ware
Background

Research
Question
Research Question /
Methodology

DTP / Agda
Big picture
Agda

Π-Ware
Syntax
Semantics
Proofs

Conclusions
Limitations
Future work

Universiteit Utrecht

Causal sequential simulation

▶ Core sequential simulation function:

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

$$[\![\text{Nil} \qquad\qquad]\!] c \ (w^0 \ , \_) = [\![\text{Nil}]\!]' \ w^0$$

$$[\![\text{Gate} \ g\# \qquad]\!] c \ (w^0 \ , \_) = [\![\text{Gate} \ g\#]\!]' \ w^0$$

$$[\![\text{Plug} \ p \qquad\quad]\!] c \ (w^0 \ , \_) = \text{plugOutputs} \ p \ w^0$$

$$[\![\text{DelayLoop} \ c \ \{p\} \ ]\!] c \qquad\quad = \text{take}_v \ j \circ \text{delay} \ c \ \{p\}$$

$$[\![c_1 \ \rangle\!\rangle' \ c_2 \ ]\!] c = [\![c_2 \ ]\!] c \circ \text{map}^+ \ [\![c_1 \ ]\!] c \circ \text{tails}^+$$

▶ Nil, Gate and Plug cases use combinational simulation

▶ DelayLoop calls a recursive helper (delay)

▶ Example structural case: $\_\rangle\!\rangle'\_$ (sequence)

  · Context of $[\![c_1 \ ]\!] c$ is context of the whole compound

  · Context of $[\![c_2 \ ]\!] c$ is past and present *outputs* of c1

Universiteit Utrecht

Sequential simulation

▶ We can then "run" the step-by-step function to produce a whole Stream

- Idea from "The Essence of Dataflow Programming" [Uustalu and Vene, 2005]

$\mathsf{runc'} : (\alpha \Rightarrow c\ \beta) \to (\Gamma c\ \alpha \times \mathsf{Stream}\ \alpha) \to \mathsf{Stream}\ \beta$
$\mathsf{runc'}\ f\ ((x^0, x^-), (x^1 :: x^+)) =$
$\quad f\ (x^0, x^-) :: \sharp\ \mathsf{runc'}\ f\ ((x^1, x^0 :: x^-), \flat\ x^+)$

$\mathsf{runc} : (\alpha \Rightarrow c\ \beta) \to (\mathsf{Stream}\ \alpha \to \mathsf{Stream}\ \beta)$
$\mathsf{runc}\ f\ (x^0 :: x^+) = \mathsf{runc'}\ f\ ((x^0, []), \flat\ x^+)$

▶ Obtaining the stream-based simulation function:

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

Universiteit Utrecht

Properties of circuits

▶ Tests and proofs about circuits depend on the *semantics*

- We focused on the functional simulation semantics
- Other possibilities (gate count, critical path, etc.)

▶ Very simple sample circuit to illustrate: XOR

Universiteit Utrecht

Sample circuit: XOR
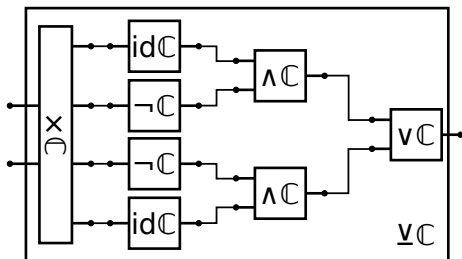
$$\underline{\vee}\mathbb{C} \;:\; \mathbb{C}\ (B \times B)\ B$$

$$\underline{\vee}\mathbb{C} \;=\; \mathsf{pFork}\!\times$$
$$\qquad \gg (\neg\mathbb{C}\ ||\ \mathsf{id}\mathbb{C} \gg \wedge\mathbb{C})\ ||\ (\mathsf{id}\mathbb{C}\ ||\ \neg\mathbb{C} \gg \wedge\mathbb{C})$$
$$\qquad \gg \vee\mathbb{C}$$

Universiteit Utrecht

Specification of XOR

Introduction
What is Π-Ware
Background

Research
Question
Research Question /
Methodology

DTP / Agda
Big picture
Agda

Π-Ware
Syntax
Semantics
Proofs

Conclusions
Limitations
Future work

▶ To define *correctness* we need a *specification function*

- Listing all possibilities (truth table)
- Based on pre-exisiting functions (standard library)

▶ Truth table

$\underline{\vee}\mathbb{C}-\text{spec}-\text{table} : (B \times B) \to B$
$\underline{\vee}\mathbb{C}-\text{spec}-\text{table} \ (false \ , \ false) = false$
$\underline{\vee}\mathbb{C}-\text{spec}-\text{table} \ (false \ , \ true \ ) = true$
$\underline{\vee}\mathbb{C}-\text{spec}-\text{table} \ (true \ \ , \ false) = true$
$\underline{\vee}\mathbb{C}-\text{spec}-\text{table} \ (true \ \ , \ true \ ) = false$

Universiteit Utrecht

Proof of XOR (truth table)

$\underline{\vee}\mathbb{C}$−proof−table : $[\![\ \underline{\vee}\mathbb{C}\ ]\!]\ (a\ ,\ b) \equiv \underline{\vee}\mathbb{C}$−spec−table $(a\ ,\ b)$
$\underline{\vee}\mathbb{C}$−proof−table false false = refl
$\underline{\vee}\mathbb{C}$−proof−table false true = refl
$\underline{\vee}\mathbb{C}$−proof−table true false = refl
$\underline{\vee}\mathbb{C}$−proof−table true true = refl

▶ Proof by *case analysis*

  • Could be automated (reflection)

Universiteit Utrecht

Specification of XOR

▶ Based (_xor_) from Data.Bool

$\_xor\_\ :\ B \rightarrow B \rightarrow B$
true xor $b = $ not b
false xor $b = b$

▶ Adapted interface to match exactly $\underline{\lor\mathbb{C}}$

$\underline{\lor\mathbb{C}}-spec-subfunc\ :\ (B \times B) \rightarrow B$
$\underline{\lor\mathbb{C}}-spec-subfunc\ =\ uncurry'\ \_xor\_$

Universiteit Utrecht

Proof of XOR (pre-existing)

▶ Proof based on $\underline{\vee}\mathbb{C}$−spec−subfunc

  $\underline{\vee}\mathbb{C}$−proof−subfunc : $[\![\ \underline{\vee}\mathbb{C}\ ]\!]\ (a\ ,\ b) \equiv \underline{\vee}\mathbb{C}$−spec−subfunc $(a\ ,\ b)$
  $\underline{\vee}\mathbb{C}$−proof−subfunc $= \underline{\vee}\mathbb{C}$−xor−equiv

▶ Need a lemma to complete the proof

  · Circuit is defined using {NOT, AND, OR}
  · _xor_ is defined directly by pattern matching

  $\underline{\vee}\mathbb{C}$−xor−equiv : $(\text{not } a \wedge b) \vee (a \wedge \text{not } b) \equiv (a \text{ xor } b)$

Universiteit Utrecht

Circuit "families"

- We can also prove properties of circuit "families"
- Example: an AND gate definition with generic number of inputs

  $\mathsf{andN}' : \forall\ n \rightarrow \mathbb{C}'\ n\ 1$
  $\mathsf{andN}'\ \mathsf{zero}\ = \top\mathbb{C}'$
  $\mathsf{andN}'\ (\mathsf{suc}\ n) = \mathsf{id}\mathbb{C}'\ |'\ \mathsf{andN}'\ n\ \ggg'\ \wedge\mathbb{C}'$

- Example proof: when all inputs are true, output is true
  - For *any* number of inputs
  - Proof by induction on n (number of inputs)

Universiteit Utrecht

Problems

- This proof is done at the *low level*

  $\text{proof}-\text{andN}' : \forall\; n \to [\![\; \text{andN}'\; n\;]\!]'\; (\text{replicate true}) \equiv [\; \text{true}\;]$
  $\text{proof}-\text{andN}'\; \text{zero}\quad = \text{refl}$
  $\text{proof}-\text{andN}'\; (\text{suc}\; n) = \text{cong}\; (\text{spec}-\text{and} \circ (\_::\_\; \text{true}))$
  $\qquad\qquad\qquad\qquad\qquad (\text{proof}-\text{andN}'\; n)$

- Still problems with inductive proofs in the high level

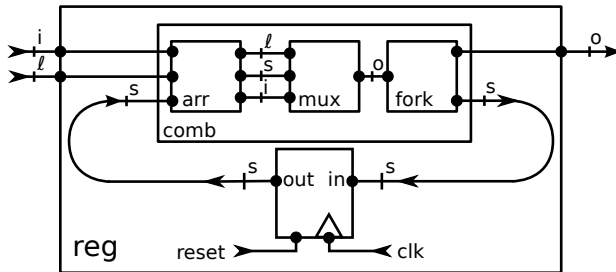 - Guess: definition of $\mathbb{C}$ and $[\![\_]\!]$ prevent goal reduction

Universiteit Utrecht

Sequential proofs

▶ Example of sequential circuit: a *register*

▶ Respective Π-Ware circuit description

$$\text{reg} : \mathbb{C} \; (B \times B) \; B$$
$$\text{reg} = \text{delay}\mathbb{C} \; (\text{arr} \; \rangle\!\rangle \; \text{mux2to1} \; \rangle\!\rangle \; \times\mathbb{C})$$
$$\quad \text{where arr} = (\Uparrow\!\Downarrow\mathbb{C} \; || \; \text{id}\mathbb{C}) \; \rangle\!\rangle \; \text{ALR}\mathbb{C} \; \rangle\!\rangle \; (\text{id}\mathbb{C} \; || \; \Uparrow\!\Downarrow\mathbb{C})$$

Universiteit Utrecht

Register example

- ▶ Example (test case) of register behaviour

  loads inputs : Stream Bool
  loads = true :: ♯ (true :: ♯ (false :: ♯ repeat false))
  inputs = true :: ♯ (false :: ♯ (true :: ♯ repeat false))

  actual = take 42 (⟦ reg ⟧∗ $ zipWith _,_ inputs loads)

  test−reg = actual ≡ true ◁ false ◁ replicate false

- ▶ Still problems with *infinite* expected vs. actual comparisons
  - Normal Agda equality (_≡_) does not work
  - Need to use *bisimilarity*

Universiteit Utrecht

What Π-Ware achieves

- ▶ Compare with Lava, Coquet
- ▶ Well-typed descriptions ($\mathbb{C}$) at *compile time*
  - Low-level descriptions ($\mathbb{C}'$) / netlists are *well-sized*

- ▶ Type safety and totality of simulation due to Agda
- ▶ Several design activities in the *same language*
  - Description (untyped / typed)
  - Simulation
  - Synthesis
  - Verification (inductive families of circuits)

Universiteit Utrecht

Current limitations / trade-offs

- Interface of generated netlists is always *flat*
  - One input, one output

```
entity fullAdd8 is
port (
    inputs  : in  std_logic_vector(16 downto 0);
    outputs : out std_logic_vector(8 downto 0)
);
end fullAdd8;
```

- Due to the indices of $\mathbb{C}'$ (naturals)
  - Can't distinguish $\mathbb{C}'$ 17 9 from $\mathbb{C}'$ $(1 + 8 + 8)$ $(8 + 1)$

Universiteit Utrecht

Current limitations / trade-offs

- Proofs on high-level families of circuits
  - Probably due to definitions of $\mathbb{C}$ and $[\![\_]\!]$
- Proofs with infinite comparisons (sequential circuits)

Universiteit Utrecht

Future work

▶ Automatic proof by reflection for finite cases
▶ Prove properties of combinators in Agda
  ▪ Algebraic properties

▶ Automatic generation of W (Synthesizable) instances
▶ More (higher) layers of abstraction

Universiteit Utrecht

Thank you!

Questions?

Mede mogelijk gemaakt door:

Utrechts
Universiteitsfonds

Universiteit Utrecht

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.

📄 Uustalu, T. and Vene, V. (2005).
The essence of dataflow programming.
In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 2–18, Berlin, Heidelberg. Springer-Verlag.

Universiteit Utrecht

References II

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