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

What is Π-Ware

What is Π-Ware

"*Π-Ware* is a Domain-Specific Language (DSL) for hardware, embedded in the dependently-typed *Agda* programming language. It allows for the description, simulation, synthesis and verification of circuits, all in the same language."

Background

Universiteit Utrecht

Hardware design is hard(er)

- Strict(er) correctness requirements
  - You can't simply *update* a full-custom chip after production
    - Intel Pentium's `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

Universiteit Utrecht

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 DSLs
  - For example, Lava (Haskell) [Bjesse et al., 1998]

Embedded DSLs for Hardware

- Lava
  - Simulation / Synthesis / Verification
  - Limitations: almost untyped / no *size checks*

```
adder :: (Signal Bool, ([Signal Bool], [Signal Bool]))
      -> ([Signal Bool], Signal Bool)
```

- Others:
  - ForSyDe [Sander and Jantsch, 1999]
  - Hawk [Launchbury et al., 1999], etc.

Dependently-Typed Programming

- Dependent type systems: systems in which types can *depend on values*
- It makes a big difference:
  - More expressivity
  - *Certified programming*
- DTP often touted as "sucessor" of functional programming
  - Very well-suited for DSLs [Oury and Swierstra, 2008]

Research Question / Methodology

Research Question / Methodology

- **Question:**
  - What are the improvements that Dependently-Typed Programming (DTP) can bring to hardware design?
    - Compared to other functional hardware languages
- **Methodology:**
  - Develop a hardware DSL, *embedded* in a dependently-typed language (Agda)
    - Called **Π-Ware**
    - Allowing simulation, synthesis and verification

Big picture

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

Dependently-Typed Programming

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

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

  - This does not:

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

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:

```
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
```

Agda

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
  - _+_ : ℕ → ℕ → ℕ defined in Data.Nat
  - if_then_else_ : Bool → $\alpha \to \alpha \to \alpha$ 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: $\varepsilon : \{\alpha : \mathsf{Set}\} \rightarrow \mathsf{Vec}\ \alpha\ \mathsf{zero}$
- "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

Circuit Syntax

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
```

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
  - 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→B = λ { False# → false;  True#  → true }
B→n = λ { false  → False#; true   → True# }
```

- Proof that n→B and B→n are inverses

```
inv−left−B   = λ { False# → refl;  True#  → refl; }
inv−right−B  = λ { false  → refl;  true   → refl }
```

- With all pieces at hand, we construct the instance

```
Atomic−B = record { Atom      = B
                  ; |Atom|−1  = |B|−1
                  ; n→atom    = n→B
                  ; atom→n    = B→n
                  ; inv−left  = inv−left−B
                  ; inv−right = 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

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|
```

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

$$
\begin{aligned}
|\mathsf{in}|\ \mathsf{FalseConst\#} &= 0\\
|\mathsf{in}|\ \mathsf{TrueConst\#} &= 0\\
|\mathsf{in}|\ \mathsf{Not\#} &= 1\\
|\mathsf{in}|\ \mathsf{And\#} &= 2\\
|\mathsf{in}|\ \mathsf{Or\#} &= 2
\end{aligned}
$$

$$|\mathsf{out}|\ \_ = 1$$

- And the specification function for each gate

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

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 }
```

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

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

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

Synthesizable

- $\Downarrow\mathsf{W}\Uparrow$ 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
```

$\Downarrow W \Uparrow$ instances

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

$$
\begin{array}{l}
\Downarrow W \Uparrow\text{-}\times : \ \{\!\!\{\ s\alpha : \Downarrow W \Uparrow\ \alpha\ \{i\}\ \}\!\!\}\ \{\!\!\{\ s\beta : \Downarrow W \Uparrow\ \beta\ \{j\}\ \}\!\!\} \\
\qquad\qquad \to \Downarrow W \Uparrow\ (\alpha \times \beta) \\
\Downarrow W \Uparrow\text{-}\times\ \{\!\!\{\ s\alpha\ \}\!\!\}\ \{\!\!\{\ s\beta\ \}\!\!\} = \Downarrow W \Uparrow[\ \mathsf{down}\ ,\ \mathsf{up}\ ] \\
\quad \mathsf{where}\ \ \mathsf{down} : (\alpha \times \beta) \to \mathsf{W}\ (i + j) \\
\qquad\qquad\ \mathsf{down}\ (a\ ,\ b) = (\Downarrow a) \mathbin{++} (\Downarrow b) \\
\\
\qquad\qquad\ \mathsf{up} : \mathsf{W}\ (i + j) \to (\alpha \times \beta) \\
\qquad\qquad\ \mathsf{up}\ w\ \mathsf{with}\ \mathsf{splitAt}\ i\ w \\
\qquad\qquad\ \mathsf{up}\ .(\Downarrow a \mathbin{++} \Downarrow b)\ |\ \Downarrow a\ ,\ \Downarrow b\ ,\ \mathsf{refl} = \Uparrow \Downarrow a\ ,\ \Uparrow \Downarrow b
\end{array}
$$

Synthesizable

- Both fields $\Downarrow$ and $\Uparrow$ should be inverses of each other
  - Due to how high-level simulation is defined using $\Downarrow$ and $\Uparrow$
- Not enforced as a field of of $\Downarrow W \Uparrow$
  - Too big of a proof burden while quick prototyping

Semantics

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

Synthesis semantics

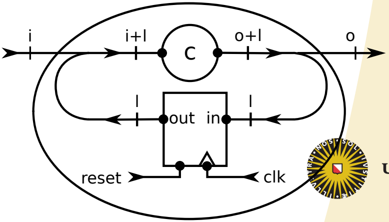
- Netlist: digraph with *gates* as nodes and *buses* as edges
- Synthesis semantics: given netlists of subcircuits, build combination

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

$$\frac{\mathsf{i}\ \mathsf{o} : \mathbb{N} \qquad \mathsf{f} : \mathsf{Fin}\ \mathsf{o} \to \mathsf{Fin}\ \mathsf{i}}{\mathsf{Plug}\ \mathsf{f} : \mathbb{C}\ \mathsf{i}\ \mathsf{o}}$$

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

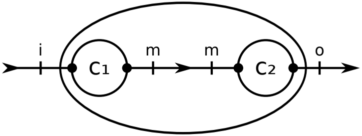
$$\frac{\mathsf{c} : \mathbb{C}\ (\mathsf{i+l})\ (\mathsf{o+l})}{\mathsf{DelayLoop} : \mathbb{C}\ \mathsf{i}\ \mathsf{o}}$$



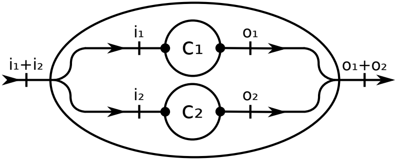
Universiteit Utrecht

Synthesis semantics

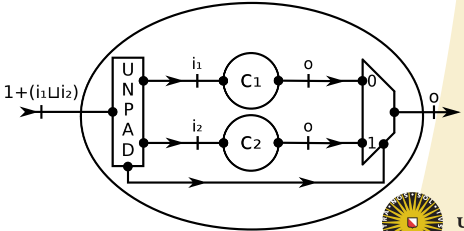
$$\frac{\begin{array}{c} c_1 : \mathbb{C}\ i\ m \\ c_2 : \mathbb{C}\ m\ o \end{array}}{c_1 \rangle\!\rangle' c_2 : \mathbb{C}\ i\ o}$$



$$\frac{\begin{array}{c} c_1 : \mathbb{C}\ i_1\ o_1 \\ c_2 : \mathbb{C}\ i_2\ o_2 \end{array}}{c_1 \mid' c_2 : \mathbb{C}\ (i_1 + i_2)\ (o_1 + o_2)}$$



$$\frac{\begin{array}{c} c_1 : \mathbb{C}\ i_1\ o \\ c_2 : \mathbb{C}\ i_2\ o \end{array}}{c_1 \mid\!+' c_2 : \mathbb{C}\ (1 + (i_1 \sqcup i_2))\ o}$$



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

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

Combinational simulation (excerpt)

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

$$\llbracket\ \mathsf{DelayLoop}\ c\ \rrbracket'\ \{()\}\ v$$

$$\llbracket\ c_1 \rangle\!\rangle' c_2\ \rrbracket'\ \{p_1\ ,\ p_2\} = \llbracket\ c_2\ \rrbracket'\ \{p_2\} \circ \llbracket\ c_1\ \rrbracket'\ \{p_1\}$$

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

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

Sequential simulation

- Inputs and outputs become Streams
  - $\mathbb{C}'\ i\ o \Longrightarrow \mathsf{Stream}\ (\mathsf{W}\ i) \rightarrow \mathsf{Stream}\ (\mathsf{W}\ o)$
  - Stream: infinite list
- We can't write a recursive evaluation function over Streams
  - *Sum* case (_|+′_) needs a function of type $(\mathsf{Stream}\ (\alpha \uplus \beta) \rightarrow \mathsf{Stream}\ \alpha \times \mathsf{Stream}\ \beta)$
    - What if there are no *lefts* (or *rights*)?
- A stream function is not an accurate model for hardware
  - A function of type $(\mathsf{Stream}\ \alpha \rightarrow \mathsf{Stream}\ \beta)$ can "look ahead"
  - For example, $\mathsf{tail}\ (x_0 :: x_1 :: x_2 :: xs) = x_1 :: x_2 :: xs$

Causal stream functions

Solution: sequential simulation based on *causal* stream function

Some definitions:

- Causal context: past + present values

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

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

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

Causal sequential simulation

- Core sequential simulation function:

$$
\begin{array}{llll}
[\![\_]\!]\mathsf{c} : \{i\ o : \mathbb{N}\} \to \mathbb{C}'\ i\ o \to (\mathsf{W}\ i \Rightarrow\mathsf{c}\ \mathsf{W}\ o) & & & \\
[\![\ \mathsf{Nil} &]\!]\mathsf{c}\ \ (w^0\ ,\ \_) & = & [\![\ \mathsf{Nil}\ ]\!]'\ w^0 \\
[\![\ \mathsf{Gate}\ g\# &]\!]\mathsf{c}\ \ (w^0\ ,\ \_) & = & [\![\ \mathsf{Gate}\ g\#\ ]\!]'\ w^0 \\
[\![\ \mathsf{Plug}\ p &]\!]\mathsf{c}\ \ (w^0\ ,\ \_) & = & \mathsf{plugOutputs}\ p\ w^0 \\
[\![\ \mathsf{DelayLoop}\ c\ \{p\} &]\!]\mathsf{c} & = & \mathsf{take}_{\mathsf{v}}\ j \circ \mathsf{delay}\ c\ \{p\}
\end{array}
$$

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

- Nil, Gate and Plug cases use combinational simulation
- DelayLoop calls a recursive helper (delay)
- Example structural case: $\_\rangle\!\rangle'\_$ (sequence)
  - Context of $[\![\ c_1\ ]\!]\mathsf{c}$ is context of the whole compound
  - Context of $[\![\ c_2\ ]\!]\mathsf{c}$ is past and present *outputs* of c1

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]

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

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

- Obtaining the stream-based simulation function:

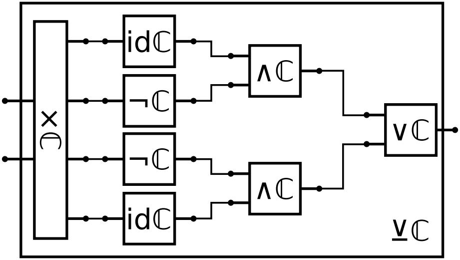
$$[\![\_]\!]{*}' : \forall\ \{i\ o\} \to \mathbb{C}'\ i\ o \to (\mathsf{Stream}\ (\mathsf{W}\ i) \to \mathsf{Stream}\ (\mathsf{W}\ o))$$
$$[\![\_]\!]{*}' = \mathsf{runc} \circ [\![\_]\!]\mathsf{c}$$

Proofs

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`

Sample circuit: XOR




```
⊻ℂ : ℂ (B × B) B
⊻ℂ =  pFork×
      ⟫ (¬ℂ || idℂ ⟫ ∧ℂ) || (idℂ || ¬ℂ ⟫ ∧ℂ)
      ⟫ ∨ℂ
```

Specification of XOR

- To define *correctness* we need a *specification function*
  - Listing all possibilities (truth table)
  - Based on pre-exisiting functions (standard library)
- Truth table

$$
\begin{aligned}
&\underline{\vee}\mathbb{C}\text{−spec−table} : (\mathrm{B} \times \mathrm{B}) \to \mathrm{B}\\
&\underline{\vee}\mathbb{C}\text{−spec−table}\ (\mathsf{false}\ ,\ \mathsf{false}) = \mathsf{false}\\
&\underline{\vee}\mathbb{C}\text{−spec−table}\ (\mathsf{false}\ ,\ \mathsf{true}\ ) = \mathsf{true}\\
&\underline{\vee}\mathbb{C}\text{−spec−table}\ (\mathsf{true}\ \ ,\ \mathsf{false}) = \mathsf{true}\\
&\underline{\vee}\mathbb{C}\text{−spec−table}\ (\mathsf{true}\ \ ,\ \mathsf{true}\ ) = \mathsf{false}
\end{aligned}
$$

Universiteit Utrecht

Proof of XOR (truth table)

```
⊻ℂ-proof-table : ⟦ ⊻ℂ ⟧ (a , b) ≡ ⊻ℂ-spec-table (a , b)
⊻ℂ-proof-table false false = refl
⊻ℂ-proof-table false true  = refl
⊻ℂ-proof-table true  false = refl
⊻ℂ-proof-table true  true  = refl
```

- Proof by *case analysis*
  - Can probably be automated by *reflection* [van der Walt and Swierstra, 2013]

Specification of XOR

- Based (_xor_) from Data.Bool

```
_xor_ : B → B → B
true  xor b = not b
false xor b = b
```

- Adapted interface to match exactly ⊻ℂ

```
⊻ℂ-spec-subfunc : (B × B) → B
⊻ℂ-spec-subfunc = uncurry′ _xor_
```

Proof of XOR (pre-existing)

- Proof based on ⊻ℂ−spec−subfunc

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

- Need a lemma to complete the proof
  - Circuit is defined using $\{\mathtt{NOT}, \mathtt{AND}, \mathtt{OR}\}$
  - _xor_ is defined directly by pattern matching

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

Circuit "families"

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

$$
\begin{aligned}
&\mathsf{andN}' \ : \forall\, n \to \mathbb{C}'\ n\ 1\\
&\mathsf{andN}'\ \mathsf{zero} \ = \mathsf{T}\mathbb{C}'\\
&\mathsf{andN}'\ (\mathsf{suc}\ n) = \mathsf{id}\mathbb{C}'\ |'\ \mathsf{andN}'\ n\ \rangle\!\rangle'\ \wedge\mathbb{C}'
\end{aligned}
$$

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

Problems

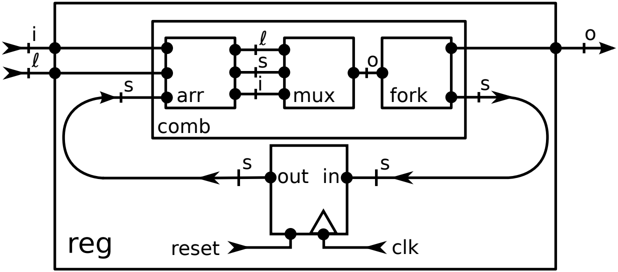
- This proof is done at the *low level*

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

- Still problems with inductive proofs in the high level
  - Guess: definition of $\mathbb{C}$ and $[\![\_]\!]$ prevent goal reduction

Sequential proofs

- Example of sequential circuit: a *register*



- Respective Π-Ware circuit description

```
reg : ℂ (B × B) B
reg = delayℂ (arr ⟫ mux2to1 ⟫ ×ℂ)
   where arr = (⇅ℂ || idℂ) ⟫ ALRℂ ⟫ (idℂ || ⇅ℂ)
```

Register example

- Example (test case) of register behaviour

$$\begin{aligned}
&\mathsf{loads\ inputs} : \mathsf{Stream\ Bool}\\
&\mathsf{loads} \;\; = \mathsf{true} :: \sharp\, (\mathsf{true} :: \sharp\, (\mathsf{false} :: \sharp\, \mathsf{repeat\ false}))\\
&\mathsf{inputs} = \mathsf{true} :: \sharp\, (\mathsf{false} :: \sharp\, (\mathsf{true} :: \sharp\, \mathsf{repeat\ false}))\\
\\
&\mathsf{actual} = \mathsf{take}\ 42\ (\llbracket\ \mathsf{reg}\ \rrbracket * \$\ \mathsf{zipWith}\ _,_\ \mathsf{inputs\ loads})\\
\\
&\mathsf{test{-}reg} = \mathsf{actual} \equiv \mathsf{true} \lhd \mathsf{false} \lhd \mathsf{replicate\ false}
\end{aligned}$$

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

Limitations

Universiteit Utrecht

What Π-Ware achieves

- Compare with Lava, Coquet
- Several design activities in the *same language*
  - Description (untyped / typed)
  - Simulation
  - Synthesis
  - Verification (inductive families of circuits)
- 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

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 $\mathbb{C}'\ (1 + 8 + 8)\ (8 + 1)$ from $\mathbb{C}'\ 17\ 9$

Current limitations / trade-offs

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

Future work

Universiteit Utrecht

Future work

- Automatic proof by reflection for finite cases
- Prove properties of combinators in Agda
  - Algebraic properties
- Automatic generation of $\Downarrow W \Uparrow$ (`Synthesizable`) instances
- More (higher) layers of abstraction

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.

Launchbury, J., Lewis, J. R., and Cook, B. (1999).
On embedding a microarchitectural design language within haskell.
SIGPLAN Not., 34(9):60–69.

Oury, N. and Swierstra, W. (2008).
The power of pi.
SIGPLAN Not., 43(9):39–50.

References II

Sander, I. and Jantsch, A. (1999).
Formal system design based on the synchrony hypothesis, functional models, and skeletons.
In *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pages 318–323. IEEE.

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.

References III

van der Walt, P. and Swierstra, W. (2013).
Engineering proof by reflection in Agda.
In *Implementation and Application of Functional Languages*, pages 157–173. Springer.

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