# 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

# What is Π-Ware

- Π-Ware är en...

# Hardware Design

## 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

# Functional Hardware

## 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, $\mu$FP [Sheeran, 1984]
- Later, *embedded* hardware Domain-Specific Languages (DSLs)
  - For example, Lava (Haskell) [Bjesse et al., 1998]

# DTP

## Dependently-Typed Programming

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

# Question

## Research Question

"What are the improvements that DTP can bring to hardware design?"

# Method

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

# Dependently-Typed Programming

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

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

  - This does not:

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



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 $\_::\_$

# 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



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
  - _+_ : ℕ → ℕ → ℕ defined in Data.Nat
  - if_then_else_ : Bool → $\alpha$ → $\alpha$ → $\alpha$ defined in Data.Bool

# 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*

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

# 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
  - 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
```

# 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  }
```

# 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|
```



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)))
```

# 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

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

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

# Synthesizable

- $\Downarrow$W$\Uparrow$ type class (pronounced `Synthesizable`)
  - Describes how to synthesize a given Agda type ($\alpha$)
  - Two fields: from element of $\alpha$ 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 (_×_)

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

# Synthesizable

- Both fields $\Downarrow$ and $\Uparrow$ should be inverses of each other

# 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

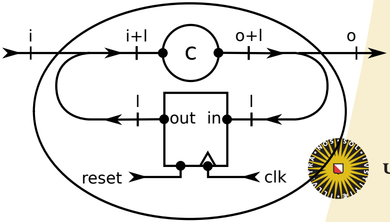
- Netlist: digraph with *gates* as nodes and *buses* as edges
- Synthesis semantics: given netlists of subcircuits, build combination

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


$$\frac{\mathrm{i}\ \mathrm{o} : \mathbb{N} \qquad \mathrm{f} : \mathrm{Fin}\ \mathrm{o} \rightarrow \mathrm{Fin}\ \mathrm{i}}{\mathrm{Plug}\ \mathrm{f} : \mathbb{C}\ \mathrm{i}\ \mathrm{o}}$$

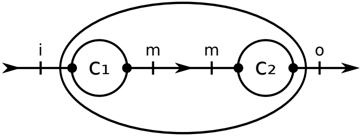

$$\frac{\mathrm{g\#} : \mathrm{Gate\#}}{\mathrm{Gate}\ \mathrm{g\#} : \mathbb{C}\ (\mathrm{ins}\ \mathrm{g\#})\ (\mathrm{outs}\ \mathrm{g\#})}$$


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

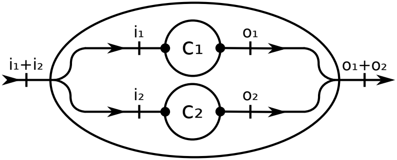


# Synthesis semantics

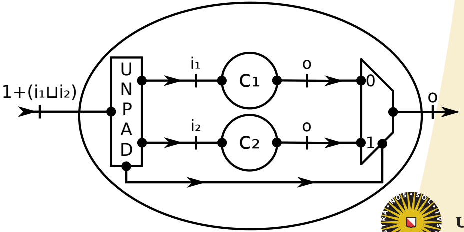
$$\frac{\begin{array}{c} c_1 : \mathbb{C}\ i\ m \\ c_2 : \mathbb{C}\ m\ o \end{array}}{c_1 \rangle\!\rangle' \ c_2 : \mathbb{C}\ i\ o}$$




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




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



# 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

# 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)

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

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

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

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

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

# Sequential simulation

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

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

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

- 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:

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

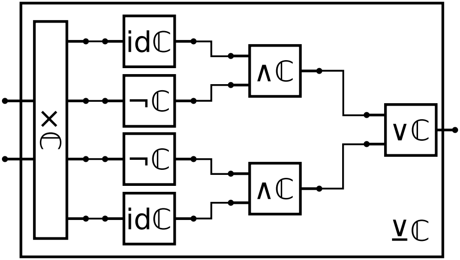


Universiteit Utrecht

# 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`

# 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

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

# 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*
  - Could be automated (reflection)

# 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 $\underline{\vee}\mathbb{C}$–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 $\{\texttt{NOT}, \texttt{AND}, \texttt{OR}\}$
  - _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

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

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



Universiteit Utrecht

# Problems

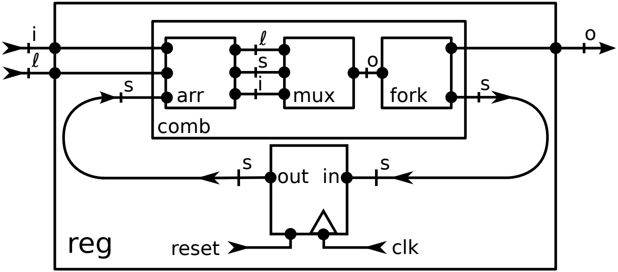
- This proof is done at the *low level*

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

- 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

$$\mathsf{reg} : \mathbb{C}\ (\mathsf{B} \times \mathsf{B})\ \mathsf{B}$$
$$\mathsf{reg} = \mathsf{delay}\mathbb{C}\ (\mathsf{arr} \gg \mathsf{mux2to1} \gg \times\mathbb{C})$$
$$\quad \mathbf{where}\ \mathsf{arr} = (\updownarrow\mathbb{C} \parallel \mathsf{id}\mathbb{C}) \gg \mathsf{ALR}\mathbb{C} \gg (\mathsf{id}\mathbb{C} \parallel \updownarrow\mathbb{C})$$

# 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*

# 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)

# 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

# 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