

# $\Pi$ -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 26<sup>th</sup> August, 2014

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# Table of Contents

## Introduction

## Research Question

## DTP / Agda

## Big picture

# Agda

Π-Ware

## Syntax

## Semantics

## Proofs

## Conclusions

## Limitations

## Future work

## Introduction

### Research Question

DTP / Agda

## Big picture

Agda

Π-Ware

## Syntax

## Semantics

## Proofs

## Conclusions

### Limitations

### Future work



Universiteit Utrecht

# What is $\Pi$ -Ware

►  $\Pi$ -Ware är en...

## Introduction

## Research Question

DTP / Agda

- Big picture
- Agda

## Π-Ware

- Syntax
- Semantics
- Proofs

## Conclusions

Limitations

Future work



Universiteit Utrecht

# Hardware Design

## Hardware design is hard(er)

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

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# 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]

## Introduction

### Research Question

DTP / Agda

## Big picture

Agda

П-Ware

### Syntax

## Semantics

## Proofs

## Conclusions

### Limitations

### Future work



Universiteit Utrecht

## Dependently-Typed Programming

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

### Introduction

Research  
Question

### DTP / Agda

Big picture  
Agda

### $\Pi$ -Ware

Syntax  
Semantics  
Proofs

### Conclusions

Limitations  
Future work



## Question

## Research Question

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

## Introduction

## Research Question

DTP / Agda

## Big picture

Agda

П-Ware

## Syntax

## Semantics

## Proofs

## Conclusions

### Limitations

### Future work



Universiteit Utrecht

# Method

## Methodology

- ▶ Develop a hardware DSL, *embedded* in a dependently-typed language (Agda)
  - Called  **$\Pi$ -Ware**
  - allowing simulation, synthesis and verification

Introduction

Research  
Question

DTP / Agda

Big picture

Agda

$\Pi$ -Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht



Limitations

Future work

# Dependently-Typed Programming

- ▶ Type checking requires *evaluation* of functions
  - We want `Vec Bool (2 + 2)` to unify with `Vec Bool 4`
- ▶ Consequence: all functions must be *total*
- ▶ Termination checker ensures (heuristics)
  - Structurally-decreasing recursion
    - This passes the check:  
`add :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$`   
`add zero y = y`  
`add (suc x') y = suc (add x' y)`
    - This does not:  
`silly :  $\mathbb{N} \rightarrow \mathbb{N}$`   
`silly zero = zero`  
`silly (suc n') = silly [ n' /2 ]`

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# Dependently-Typed Programming

- ▶ Dependent pattern matching can *rule out* impossible cases

## Introduction

## Research Question

DTP / Agda

## Big picture

Agda

## Π-Ware

## Syntax

## Semantics

## Proofs

## Conclusions

### Limitations

### Future work



Universiteit Utrecht

# Dependently-Typed Programming

- ▶ Dependent pattern matching can *rule out* impossible cases

- Classic example: *safe head* function

$$\text{head} : \text{Vec } \alpha \ (\text{suc } n) \rightarrow \alpha$$
$$\text{head } (x :: xs) = x$$

## Introduction

## Research Question

DTP / Agda

Big picture  
Agda

## Π-Ware

- Syntax
- Semantics
- Proofs

## Conclusions

Limitations

Future work



Universiteit Utrecht

# Dependently-Typed Programming

- ▶ Dependent pattern matching can *rule out* impossible cases
  - Classic example: *safe head* function
$$\text{head} : \text{Vec } \alpha \ (\text{suc } n) \rightarrow \alpha$$
$$\text{head } (x :: xs) = x$$
  - The **only** constructor returning  $\text{Vec } \alpha \ (\text{suc } n)$  is `__::__`

## Research Question

- Big picture
- Agda

- Syntax
- Semantics
- Proofs

Limitations

Future work



# Dependent types as logic

- ▶ Programming language / Theorem prover
  - Types as propositions, terms as proofs [Wadler, 2014]

- ▶ Example:

- Given the relation (drawn triangle):

```
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

Research  
Question

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**)
  - $a \equiv b + c$  is a valid identifier,  $a \equiv b + c$  an expression
  - Actually used in Agda's standard library
  - And in  $\Pi$ -Ware:  $\mathbb{C}$ ,  $\llbracket c \rrbracket$ ,  $\Downarrow$ ,  $\Uparrow$
- ▶ *Mixfix* notation
  - $\_[_] := \_$  is the vector update function:  $v \ [ \ # \ 3 \ ] := \text{true}$ .
  - $\_[_] := \_ \ v \ (\# \ 3) \ \text{true} \iff v \ [ \ # \ 3 \ ] := \text{true}$
- ▶ 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`

Introduction

Research  
Question

DTP / Agda

Big picture

Agda

$\Pi$ -Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

# Agda syntax for Haskell programmers

- ▶ Implicit arguments

- Don't have to be passed if Agda can **guess** it
- Syntax:  $\varepsilon : \{ \alpha : \text{Set} \} \rightarrow \text{Vec } \alpha \text{ zero}$

► “For all” syntax:  $\forall n \iff (n : \_)$

- Where `_` means: guess this type (based on other args)
- Example:
  - $\forall n \rightarrow \text{zero} \leq n$
  - `data < : ℕ → ℕ → Set`

- ▶ It's common to combine both:

- $\forall \{ \alpha \ n \} \rightarrow \text{Vec } \alpha \ (\text{succ } n) \rightarrow \alpha \iff$   
 $\{ \alpha : \ \ \ \ \} \{ n : \ \ \ \} \rightarrow \text{Vec } \alpha \ n \rightarrow \alpha$

## Introduction

## Research Question

DTP / Agda

## Big picture

Agda

П-Ware

## Syntax

## Semantics

## Proofs

## Conclusions

### Limitations

### Future work



Universiteit Utrecht



# Low-level circuits

- Structural representation
- Untyped but *sized*

data  $\mathbb{C}' : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set}$

data  $\mathbb{C}'$  where

Nil :  $\mathbb{C}'$  zero zero

Gate :  $(g\# : \text{Gates}\#) \rightarrow \mathbb{C}' (|\text{in}| g\#) (|\text{out}| g\#)$

Plug :  $\forall \{i\ o\} \rightarrow (f : \text{Fin } o \rightarrow \text{Fin } i) \rightarrow \mathbb{C}' i\ o$

DelayLoop :  $(c : \mathbb{C}' (i + l) (o + l)) \{\text{comb}'\ c\} \rightarrow \mathbb{C}' i\ o$

$\_ \gg' \_$  :  $\mathbb{C}' i\ m \rightarrow \mathbb{C}' m\ o \rightarrow \mathbb{C}' i\ o$

$\_ |' \_$  :  $\mathbb{C}' i_1\ o_1 \rightarrow \mathbb{C}' i_2\ o_2 \rightarrow \mathbb{C}' (i_1 + i_2) (o_1 + o_2)$

$\_ |+' \_$  :  $\mathbb{C}' i_1\ o \rightarrow \mathbb{C}' i_2\ o \rightarrow \mathbb{C}' (\text{succ } (i_1 \sqcup i_2))\ o$

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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|
```

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

Π-Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# Atomic instances

- ▶ Examples of types that can be **Atomic**
  - Bool, std\_logic, other multi-valued logics
  - Predefined in the library: **PiWare.Atom.Bool**
- ▶ First, define how many atoms we are interested in

**|B|** - 1 = 1

**|B|** = suc **|B|** - 1

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

pattern **False#** = Fz

pattern **True#** = Fs Fz

Introduction

Research  
Question

DTP / Agda

Big picture

Agda

**Π**-Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

# Atomic instance (Bool)

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

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

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

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

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

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

- With all pieces at hand, we construct the instance

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

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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](#)

## Introduction

## Research Question

DTP / Agda

## Big picture

Agda

Π-Ware

### Syntax

## Semantics

## Proofs

## Conclusions

### Limitations

### Future work



Universiteit Utrecht

# The Gates type class

$W : \mathbb{N} \rightarrow \text{Set}$

$W = \text{Vec Atom}$

record Gates : Set where

field

|Gates| :  $\mathbb{N}$

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

spec : (g : Fin |Gates|)  
 $\rightarrow (W (|in| g) \rightarrow W (|out| g))$

Gates# = Fin |Gates|

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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)))
```

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# Gates instance (BoolTrio)

- Defining the *interfaces* of the gates

|in| FalseConst# = 0

|in| TrueConst# = 0

|in| Not# = 1

|in| And# = 2

|in| Or# = 2

|out| \_ = 1

- And the specification function for each gate

spec-false \_ = [ false ]

spec-true \_ = [ true ]

spec-not (x :: ε) = [ not x ]

spec-and (x :: y :: ε) = [ x ∧ y ]

spec-or (x :: y :: ε) = [ x ∨ y ]

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

Π-Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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 }
```

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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
  - The input and output indices are Agda 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}
```

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

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# 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  $\Downarrow W \Uparrow$  ( $\alpha$  : Set) { $i$  :  $\mathbb{N}$ } : Set where
  constructor  $\Downarrow W \Uparrow$  [ $\_$ ,  $\_$ ]
  field
```

$$\Downarrow : \alpha \rightarrow W\ i$$
$$\Uparrow : W\ i \rightarrow \alpha$$

Introduction

Research  
Question

DTP / Agda

Big picture

Agda

$\Pi$ -Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

## $\Downarrow W \Uparrow$ instances

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

$$\Downarrow W \Uparrow - \times : \{ \mid s\alpha : \Downarrow W \Uparrow \alpha \{i\} \} \{ \mid s\beta : \Downarrow W \Uparrow \beta \{j\} \} \\ \rightarrow \Downarrow W \Uparrow (\alpha \times \beta)$$

$$\Downarrow W \Uparrow - \times \{ \alpha \} \{ i \} \{ \beta \} \{ j \} \{ \mid s\alpha \} \{ \mid s\beta \} = \Downarrow W \Uparrow [ \text{down} , \text{up} ]$$

where  $\text{down} : (\alpha \times \beta) \rightarrow W (i + j)$   
 $\text{down} (a , b) = (\Downarrow a) ++ (\Downarrow b)$

$$\text{up} : W (i + j) \rightarrow (\alpha \times \beta)$$

$$\text{up } w \text{ with splitAt } i \text{ } w$$

$$\text{up } .(\Downarrow a ++ \Downarrow b) \mid \Downarrow a , \Downarrow b , \text{refl} = \Uparrow \Downarrow a , \Uparrow \Downarrow b$$

Introduction

Research  
Question

DTP / Agda

Big picture

Agda

$\Pi$ -Ware

Syntax

Semantics  
Proof

Conclusions

Limitations

Future work



Universiteit Utrecht

# Synthesizable

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

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht



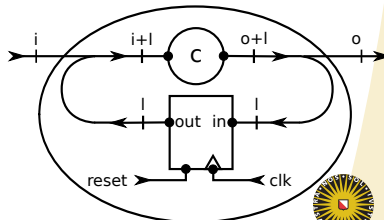
# Synthesis semantics

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

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

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

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

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


Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

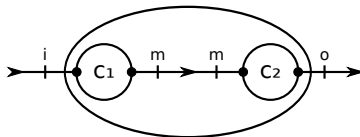
Limitations  
Future work



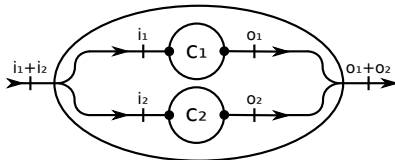
Universiteit Utrecht

# Synthesis semantics

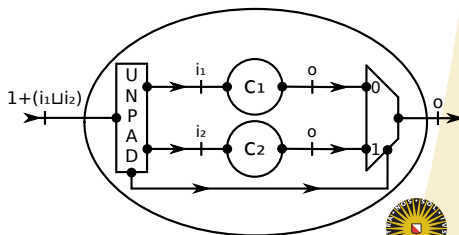
$$\frac{c_1 : \mathbb{C} \, i \, m \quad c_2 : \mathbb{C} \, m \, o}{c_1 \gg' c_2 : \mathbb{C} \, i \, o}$$



$$\frac{c_1 : \mathbb{C} \, i_1 \, o_1 \quad c_2 : \mathbb{C} \, i_2 \, o_2}{c_1 \mid' c_2 : \mathbb{C} \, (i_1 + i_2) \, (o_1 + o_2)}$$



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



Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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 **VHDL**Entity
- **netlist** :  $(g\# : \text{Gates}\#) \rightarrow \text{VHDL} \text{Entity} (|in| \ g\#) (|out| \ g\#)$ 
  - The VHDL entity has the *interface* of corresponding gate

## Introduction

## Research Question

DTP / Agda

## Big picture

Agda

П-Ware

### Syntax

## Semantics

## Proofs

## Conclusions

### Limitations

### Future work



Universiteit Utrecht

## Simulation semantics

- ▶ Two levels of abstraction
  - High-level simulation ( $\llbracket \_ \rrbracket$ ) for high-level circuits ( $\mathbb{C}$ )
  - Low-level simulation ( $\llbracket \_ \rrbracket'$ ) for low-level circuits ( $\mathbb{C}'$ )
- ▶ Two kinds of simulation
  - Combinational simulation ( $\llbracket \_ \rrbracket$ ) for stateless circuits
  - Sequential simulation ( $\llbracket \_ \rrbracket^*$ ) for stateful circuits
- ▶ High level defined in terms of low level

$$\begin{aligned} \llbracket \_ \rrbracket &: \forall \{ \alpha \ i \ \beta \ j \} \rightarrow (c : \mathbb{C} \ \alpha \ \beta \ \{i\} \ \{j\}) \rightarrow (\alpha \rightarrow \beta) \\ \llbracket \text{MkC} \ \{ \ s\alpha \} \ \{ \ s\beta \} \ c' \rrbracket &= \uparrow \circ \llbracket c' \rrbracket' \circ \downarrow \end{aligned}$$

Limitations

Future work



Universiteit Utrecht

## Combinational simulation (excerpt)

$$\llbracket \_ \rrbracket' : \forall \{i\ o\} \rightarrow (c : \mathbb{C}'\ i\ o) \{p : \text{comb}'\ c\} \rightarrow (\mathbb{W}\ i \rightarrow \mathbb{W}\ o)$$
$$[\text{Ni}]' = \text{const } \varepsilon$$
$$\llbracket \text{Gate } g^\# \rrbracket' = \text{spec } g^\#$$
$$[\![ \text{Plug } p \ ]\!] = \text{plugOutputs } p$$
$$\llbracket \text{DelayLoop } c \rrbracket' \{ () \} v$$
$$\llbracket c_1 \gg' c_2 \rrbracket' \{p_1, p_2\} = \llbracket c_2 \rrbracket' \{p_2\} \circ \llbracket c_1 \rrbracket' \{p_1\}$$
$$\llbracket \_+ \_ - \{i_1\} \ c_1 \ c_2 \rrbracket' \{p_1, p_2\} =$$

$$\llbracket \llbracket c_1 \rrbracket' \{p_1\}, \llbracket c_2 \rrbracket' \{p_2\} \rrbracket' \circ \text{untag} \{i_1\}$$

► Remarks:

- Proof required that  $c$  is combinational
- **Gate** case uses specification function
- **DelayLoop** case can be *discharged*

## Introduction

## Research Question

DTP / Agda

- Big picture
- Agda

П-Ware

Syntax

**Semantics**

Proofs

## Conclusions

Limitations

Future work



Universiteit Utrecht

## Sequential simulation

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

## Research Question

- Big picture
- Agda

Syntax

**Semantics**

Proofs

Limitations

Future work



Universiteit Utrecht

# Causal stream functions

Solution: sequential simulation using *causal* stream function

Some definitions:

- ▶ Causal context: past + present values

$$\Gamma c : (\alpha : \text{Set}) \rightarrow \text{Set}$$

$$\Gamma c \alpha = \alpha \times \text{List } \alpha$$

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

$$\_ \Rightarrow^c \_ : (\alpha \beta : \text{Set}) \rightarrow \text{Set}$$

$$\alpha \Rightarrow^c \beta = \Gamma c \alpha \rightarrow \beta$$

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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]

$\text{runc}' : (\alpha \Rightarrow_{\mathbf{c}} \beta) \rightarrow (\Gamma_{\mathbf{c}} \alpha \times \text{Stream } \alpha) \rightarrow \text{Stream } \beta$

$\text{runc}' f ((x^0, x^-), (x^1 :: x^+)) =$   
 $f (x^0, x^-) :: \# \text{runc}' f ((x^1, x^0 :: x^-), \mathbf{b} x^+)$

$\text{runc} : (\alpha \Rightarrow_{\mathbf{c}} \beta) \rightarrow (\text{Stream } \alpha \rightarrow \text{Stream } \beta)$

$\text{runc } f (x^0 :: x^+) = \text{runc}' f ((x^0, []), \mathbf{b} x^+)$

- Obtaining the stream-based simulation function:

$\llbracket \_ \rrbracket *' : \forall \{i \ o\} \rightarrow \mathbb{C}' \ i \ o \rightarrow (\text{Stream } (\mathbf{W} \ i) \rightarrow \text{Stream } (\mathbf{W} \ o))$

$\llbracket \_ \rrbracket *' = \text{runc} \circ \llbracket \_ \rrbracket_{\mathbf{c}}$

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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

## Introduction

## Research Question

DTP / Agda

- Big picture
- Agda

Π-Ware

Syntax

Semantics

## Proofs

## Conclusions

Limitations

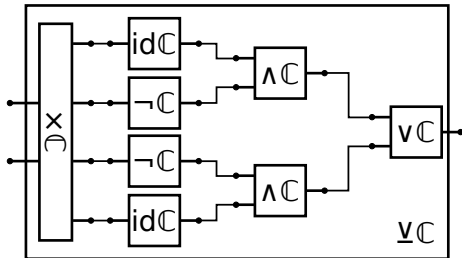
Future work



Universiteit Utrecht



# Sample circuit: XOR



$\underline{vC} : C (B \times B) B$

$\underline{vC} = \text{pForkX}$

$\gg (\neg C \parallel \text{idC} \gg \wedge C) \parallel (\text{idC} \parallel \neg C \gg \wedge C)$   
 $\gg vC$

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics

Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# Specification of XOR

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

$\underline{\text{VC}}\text{-spec-table} : (B \times B) \rightarrow B$

$\underline{\text{VC}}\text{-spec-table} \text{ (false , false) } = \text{false}$

$\underline{\text{VC}}\text{-spec-table} \text{ (false , true) } = \text{true}$

$\underline{\text{VC}}\text{-spec-table} \text{ (true , false) } = \text{true}$

$\underline{\text{VC}}\text{-spec-table} \text{ (true , true) } = \text{false}$

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# Proof of XOR (truth table)

$\underline{\vee}\mathbb{C}\text{-proof-table} : \llbracket \underline{\vee}\mathbb{C} \rrbracket (a, b) \equiv \underline{\vee}\mathbb{C}\text{-spec-table} (a, b)$

$\underline{\vee}\mathbb{C}\text{-proof-table} \text{ false false} = \text{refl}$

$\underline{\vee}\mathbb{C}\text{-proof-table} \text{ false true} = \text{refl}$

$\underline{\vee}\mathbb{C}\text{-proof-table} \text{ true false} = \text{refl}$

$\underline{\vee}\mathbb{C}\text{-proof-table} \text{ true true} = \text{refl}$

- Proof by *case analysis*
  - Could be automated (reflection)

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics

Proofs

Conclusions

Limitations  
Future work



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 `⊔`

`⊔-spec-subfunc` :  $(B \times B) \rightarrow B$

`⊔-spec-subfunc = uncurry' _xor_`

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics

Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# Proof of XOR (pre-existing)

- Proof based on `VC-spec-subfunc`

`VC-proof-subfunc` :  $\llbracket \text{VC} \rrbracket (a, b) \equiv \text{VC-spec-subfunc } (a, b)$   
`VC-proof-subfunc` = `VC-xor-equiv`

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

`VC-xor-equiv` :  $(\text{not } a \wedge b) \vee (a \wedge \text{not } b) \equiv (a \text{ xor } b)$

Introduction

Research  
Question

DTP / Agda  
Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# Circuit “families”

- ▶ We can also prove properties of circuit “families”
- ▶ Example: an AND gate with a generic number of inputs

$\text{andN}' : \forall n \rightarrow \mathbb{C}' \ n \ 1$

$\text{andN}' \ \text{zero} = \text{TC}'$

$\text{andN}' \ (\text{suc } n) = \text{idC}' \mid' \text{andN}' \ n \ \gg' \wedge \mathbb{C}'$

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

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

# Problems

- This proof is done in the *low level*

```
proof-andN' : ∀ n → [ andN' n ]' (replicate true) ≡ [ true ]
proof-andN' zero      = refl
proof-andN' (suc n) = cong (spec-and ∘ (λ_::_ true))
                      (proof-andN' n)
```

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

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

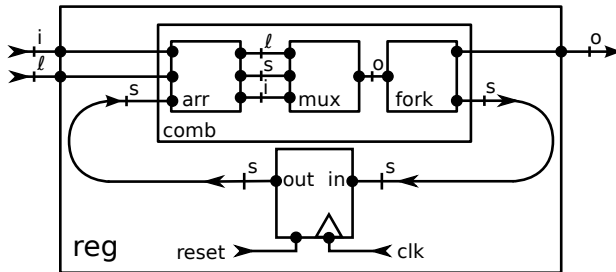
Limitations  
Future work



Universiteit Utrecht

# Sequential proofs

- Example of sequential circuit: a *register*



- Respective  $\Pi$ -Ware circuit description

$\text{reg} : \mathbb{C} (B \times B) B$

$\text{reg} = \text{delayC} (\text{arr} \gg \text{mux2to1} \gg \times C)$

where  $\text{arr} = (\uparrow C \parallel \text{idC}) \gg \text{ALRC} \gg (\text{idC} \parallel \uparrow C)$

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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  $\equiv$  true  $\triangleleft$  false  $\triangleleft$  replicate false

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

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



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)

## Introduction

## Research Question

DTP / Agda

- Big picture
- Agda

Π-Ware

- Syntax
- Semantics
- Proofs

## Conclusions

### Limitations



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

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht

## Current limitations / trade-offs

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

## Introduction

### Research Question

DTP / Agda

## Big picture

Agda

Π-Ware

### Syntax

## Semantics

## Proofs

## Conclusions

### Limitations

### Future work



Universiteit Utrecht

## Future work

- ▶ Automatic proof by reflection for finite cases
- ▶ Prove properties of combinators in Agda
- ▶ Automatic generation of  $W$  (Synthesizable) instances
- ▶ More layers of abstraction

## Introduction

### Research Question

DTP / Agda

- Big picture
- Agda

Π-Ware

- Syntax
- Semantics
- Proofs

## Conclusions

Limitations

Future work



Universiteit Utrecht

# Thank you!

## Questions?

Mede mogelijk gemaakt door:

Utrechts  
**Universiteitsfonds**



**Universiteit Utrecht**



**Universiteit Utrecht**

## References I



In *Proceedings of the Third Asian Conference on Programming Languages and Systems, APLAS'05*, pages 2–18, Berlin, Heidelberg. Springer-Verlag.

### Future work



53

# 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>.

Introduction

Research  
Question

DTP / Agda

Big picture  
Agda

$\Pi$ -Ware

Syntax  
Semantics  
Proofs

Conclusions

Limitations  
Future work



Universiteit Utrecht