

# $\Pi$ -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

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht

# What is $\Pi$ -Ware

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



# What is $\Pi$ -Ware

“ *$\Pi$ -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.”

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture  
Agda

## $\Pi$ -Ware

Circuit Syntax  
Semantics  
Proofs

## Conclusions

Summary  
Future work



Universiteit Utrecht

# Background

## Introduction

What is  $\Pi$ -Ware

**Background**

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



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 FDIIV
  - Expensive verification / validation
    - Up to 50% of development costs
- ▶ Low-level details (more) important
  - Layout / area
  - Power consumption / fault tolerance

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



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

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



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

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht

# Functional Hardware Description

- ▶ A functional program describes a circuit
  - Easier to *reason* about program properties
  - Inherently *parallel* and *stateless* semantics
- ▶ Several *functional* HDLs during the 1980s
  - For example,  $\mu$ FP [Sheeran, 1984]
- ▶ Later, *embedded* hardware DSLs
  - For example, Lava (Haskell) [Bjesse et al., 1998]

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht



# 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.

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture  
Agda

## $\Pi$ -Ware

Circuit Syntax  
Semantics  
Proofs

## Conclusions

Summary  
Future work



Universiteit Utrecht

# 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 “successor” of functional programming
  - Very well-suited for DSLs [Oury and Swierstra, 2008]

## What is Π-Ware

## Research Question

Research Question /  
Methodology

- Big picture
- Agda

- Circuit Syntax
- Semantics
- Proofs

Summary

Future work



Universiteit Utrecht

# Research Question / Methodology

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



## 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)
  - Allowing simulation, synthesis and verification

## Introduction

## What is Π-Ware

## Background

## Research Question

Research Question /  
Methodology

DTP / Agda

## Big picture

Agda

## Π-Ware

### Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### Future work



Universiteit Utrecht

# Big picture

## Introduction

What is  $\Pi$ -Ware  
Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

**Big picture**  
Agda

## $\Pi$ -Ware

Circuit Syntax  
Semantics  
Proofs

## Conclusions

Summary  
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 (heuristics)
  - Structurally-decreasing recursion
  - This passes the check:  

```
add : ℕ → ℕ → ℕ
add zero    y = y
add (suc x') y = suc (add x' y)
```
  - This does not:  

```
silly : ℕ → ℕ
silly zero    = zero
silly (suc n') = silly [ n' /2]
```

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht

# Dependently-Typed Programming

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

- ▶ Classic example: *safe* **head** function

**head** : **Vec**  $\alpha$  (**suc**  $n$ )  $\rightarrow \alpha$

**head** ( $x :: xs$ ) =  $x$

- The **only** constructor returning **Vec**  $\alpha$  (**suc**  $n$ ) is **\_\_::\_\_**

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht





# Agda

## Introduction

What is  $\Pi$ -Ware  
Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture  
**Agda**

## $\Pi$ -Ware

Circuit Syntax  
Semantics  
Proofs

## Conclusions

Summary  
Future work



# Agda syntax for Haskell programmers

- ▶ Liberal identifier lexing (Unicode **everywhere**)
  - $a \equiv b + c$  is a valid identifier,  $a \equiv b + c$  an expression
  - Used a lot in Agda's standard library:  $\times$ ,  $\uplus$ ,  $\wedge$
  - 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`
  - $\text{if\_then\_else\_} : \text{Bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$  defined in `Data.Bool`

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

## What is Π-Ware

## Background

## Research Question

Research Question /  
Methodology

DTP / Agda

## Big picture

Agda

## Π-Ware

### Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### Future work



Universiteit Utrecht

# Circuit Syntax

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

**Circuit Syntax**

Semantics

Proofs

## Conclusions

Summary

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

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

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

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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
  - Need at least 1 (later why)

$|B|-1 = 1$

$|B| = \text{suc } |B|-1$

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

**pattern** **False#** = **Fz**

**pattern** **True#** = **Fs Fz**

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

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

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

## What is Π-Ware

## Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

## Big picture

Agda

П-Ware

## Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### 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 :  $(g : \text{Fin } |Gates|) \rightarrow (W \text{ (|in| } g) \rightarrow W \text{ (|out| } g))$

Gates# = Fin |Gates|

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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  $(x :: \varepsilon)$  = [ not  $x$  ]

spec-and  $(x :: y :: \varepsilon)$  = [  $x \wedge y$  ]

spec-or  $(x :: y :: \varepsilon)$  = [  $x \vee y$  ]

## Introduction

What is  $\Pi$ -Ware

Background

## Research

## Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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
```

$$\text{specs} - \text{BoolTrio } \text{Not\#} = \text{spec} - \text{not}$$
$$\text{specs-BoolTrio } \text{And\#} = \text{spec-and}$$

```
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

## What is Π-Ware

## Background

## Research Question

Research Question / Methodology

DTP / Agda

## Big picture

Agda

Π-Ware

### Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### 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

- 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* (class constraints)

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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 (`_×_`)

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

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

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

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

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

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

### Introduction

What is  $\Pi$ -Ware

Background

### Research

### Question

Research Question /  
Methodology

### DTP / Agda

Big picture

Agda

### $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

### Conclusions

Summary

Future work



## 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  $\Downarrow W \Uparrow$ 
  - Too big of a proof burden while quick prototyping

## Introduction

## What is Π-Ware

## Background

## Research Question

Research Question / Methodology

## DTP / Agda

## Big picture

Agda

## Π-Ware

## Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### Future work



Universiteit Utrecht

# Semantics

## Introduction

What is  $\Pi$ -Ware  
Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture  
Agda

## $\Pi$ -Ware

Circuit Syntax  
**Semantics**  
Proofs

## Conclusions

Summary  
Future work





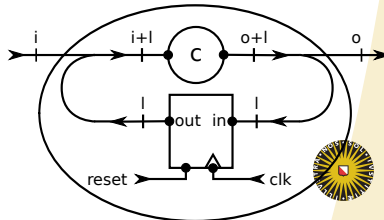
# Synthesis semantics

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

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

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

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

$$\frac{c : \mathbb{C} \ (i+1) \ (o+1)}{\text{DelayLoop} : \mathbb{C} \ i \ o}$$


## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

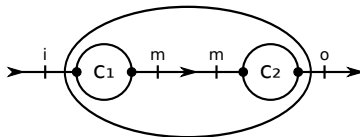
## Conclusions

Summary

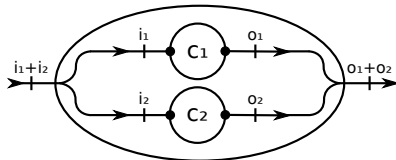
Future work

# Synthesis semantics

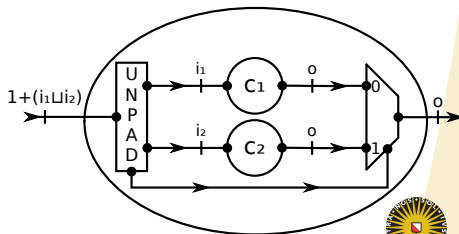
$$\frac{c_1 : \mathbb{C} \ i \ m \quad c_2 : \mathbb{C} \ m \ o}{c_1 \gg' c_2 : \mathbb{C} \ i \ o}$$



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



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



## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

## What is Π-Ware

## Background

## Research Question

Research Question /  
Methodology

DTP / Agda

## Big picture

Agda

П-Ware

### Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

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

## Introduction

## What is Π-Ware

## Background

### Research Question

Research Question / Methodology

DTP / Agda

## Big picture

Agda

Π-Ware

### Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### Future work



Universiteit Utrecht



# Causal stream functions

Solution: sequential simulation based on *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

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht

## Proofs

## Introduction

## What is Π-Ware

## Background

## Research Question

Research Question / Methodology

## DTP / Agda

## Big picture

Agda

## Π-Ware

## Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### Future work



# 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

## What is Π-Ware

## Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

## Big picture

Agda

## Π-Ware

## Circuit Syntax

## Semantics

## Proofs

## Conclusions

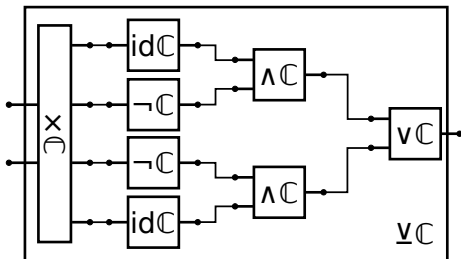
## Summary

### Future work



Universiteit Utrecht

# Sample circuit: XOR



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

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

$\gg (\neg C \parallel idC \gg \wedge C) \parallel (idC \parallel \neg C \gg \wedge C)$   
 $\gg vC$

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht

## Proof of XOR (truth table)

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

```
VC-proof-table  false  false  = refl
```

$$\text{VC-proof-table } \text{false } \text{true} = \text{refl}$$

```
VC-proof-table  true  false = refl
```

$$\text{VC-proof-table} \quad \text{true} \quad \text{true} \quad = \text{refl}$$

► Proof by *case analysis*

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

## Introduction

## What is Π-Ware

## Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

## Big picture

Agda

## Π-Ware

### Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### 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 `⋄C`

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

`⋄C-spec-subfunc` = `uncurry' _xor_`

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht

## Proof of XOR (pre-existing)

- Proof based on  $\forall C\text{-spec-subfunc}$

$$\begin{aligned} \underline{\text{VC-proof-subfunc}} &: \llbracket \underline{\text{VC}} \rrbracket (a, b) \equiv \underline{\text{VC-spec-subfunc}} (a, b) \\ \underline{\text{VC-proof-subfunc}} &= \underline{\text{VC-xor-equiv}} \end{aligned}$$

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

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

## Introduction

## What is Π-Ware

## Background

Research

### Question

**a, b)** Research Question / Methodology

DTP / Agda

## Big picture

Agda

## Π-Ware

### Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### Future work



Universiteit Utrecht

# Circuit “families”

- ▶ We can also prove properties of circuit “families”
- ▶ Example: an AND gate definition with 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 **true**, output is **true**
  - For *any* number of inputs
  - Proof by induction on  $n$  (number of inputs)

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht

# Problems

- This proof is done at the *low level*

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

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

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

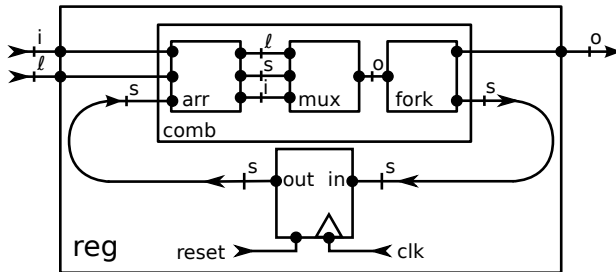
Future work



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

What is  $\Pi$ -Ware

Background

## Research

### Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

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

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht



# Summary

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

**Summary**

Future work



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  $\mathbb{C}'179$

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht

# Current limitations / trade-offs

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

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht

# Future work

## Introduction

What is  $\Pi$ -Ware  
Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

Big picture  
Agda

## $\Pi$ -Ware

Circuit Syntax  
Semantics  
Proofs

## Conclusions

Summary  
**Future work**



## 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

## Introduction

## What is Π-Ware

## Background

## Research Question

Research Question /  
Methodology

## DTP / Agda

## Big picture

Agda

## Π-Ware

### Circuit Syntax

## Semantics

## Proofs

## Conclusions

## Summary

### Future work



Universiteit Utrecht

# Thank you!

## Questions?

Mede mogelijk gemaakt door:

Utrechts  
**Universiteitsfonds**



**Universiteit Utrecht**



**Universiteit Utrecht**

## References I



*SIGPLAN Not.*, 43(9):39–50.

### Future work





# 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.

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions


Summary

Future work



Universiteit Utrecht

# 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>.

## Introduction

What is  $\Pi$ -Ware

Background

## Research Question

Research Question / Methodology

## DTP / Agda

Big picture

Agda

## $\Pi$ -Ware

Circuit Syntax

Semantics

Proofs

## Conclusions

Summary

Future work



Universiteit Utrecht