

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

Sunday 24th August, 2014

Background

Hardware Design
Functional Hardware
DTP

Research
Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



Universiteit Utrecht

Table of Contents

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

DTP / Agda

Big picture

Agda

Π -Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

DTP / Agda

Big picture

Agda

Π -Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

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 FDIV
- Expensive verification / validation (up to 50% of development costs)

- ▶ Low-level details (more) important

- Layout / area
- Power consumption / fault tolerance

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

DTP / Agda

Big picture

Agda

Π-Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

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

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

DTP / Agda

Big picture

Agda

Π -Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

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

Background

Hardware Design

Functional Hardware

DTP

Research

Question

Question

Method

DTP / Agda

Big picture

Agda

Π -Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

Functional Programming

- ▶ Easier to *reason* about program properties
- ▶ Inherently *parallel* and *stateless* semantics
 - In contrast to imperative programming

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

DTP / Agda

Big picture

Agda

Π -Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

Functional Hardware Description

- ▶ A functional program describes a circuit
- ▶ Several *functional* Hardware Description Languages (HDLs) during the 1980s
 - For example, μ FP [Sheeran, 1984]
- ▶ Later, *embedded* hardware Domain-Specific Languages (DSLs)
 - For example, Lava (Haskell) [Bjesse et al., 1998]

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

DTP / Agda

Big picture

Agda

Π-Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

Embedded DSLs for Hardware

- ▶ Lava
- ▶ Limitations
 - Low level types
 - Not guaranteeing size match

Background

Hardware Design

Functional Hardware

DTP

Research

Question

Question

Method

DTP / Agda

Big picture

Agda

Π-Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

Dependently-Typed Programming

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

Background

- Hardware Design
- Functional Hardware
- DTP**

Research Question

- Question
- Method

DTP / Agda

- Big picture
- Agda

Π -Ware

- Syntax
- Semantics
- Proofs

Conclusions

- Limitations
- Future work



Research Question

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

Background

- Hardware Design
- Functional Hardware
- DTP

Research Question

- Question
- Method

DTP / Agda

- Big picture
- Agda

Π -Ware

- Syntax
- Semantics
- Proofs

Conclusions

- Limitations
- Future work



Methodology

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

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



Dependently-Typed Programming

- ▶ Types can depend on values

- Example: `data Vec (α : Set) : ℕ → Set where...`
- Compare with Haskell (GADT style):
`data List :: * -> * where...`

- Types of arguments can depend on *values of previous arguments*

- Ensure a “safe” domain
- $\text{take} : (m : \mathbb{N}) \rightarrow \text{Vec } \alpha \rightarrow (m + n) \rightarrow \text{Vec } \alpha$

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

DTP / Agda

Big picture

Agda

П-Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

Dependently-Typed Programming

- ▶ Type checking requires *evaluation* of functions
 - We want `Vec Bool (2 + 2)` to unify with `Vec Bool 4`
- ▶ Consequence: all functions must be *total*
- ▶ Termination checker 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]`

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

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

Background

- Hardware Design
- Functional Hardware
- DTP

Research Question

- Question
- Method

DTP / Agda

- Big picture
- Agda

Π -Ware

- Syntax
- Semantics
- Proofs

Conclusions

- Limitations
- Future work



Dependently-Typed Programming

- ▶ Dependent pattern matching can *rule out* impossible cases
 - Classic example: *safe head* function

$$\begin{aligned} \text{head} &: \text{Vec } \alpha \ (\text{succ } n) \rightarrow \alpha \\ \text{head } (x :: xs) &= x \end{aligned}$$

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

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 $_::_$

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

DTP / Agda

Big picture

Agda

Π-Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

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
```

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π-Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
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
 - Actually used in Agda's standard library
 - And in Π -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**

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -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:

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

Background

- Hardware Design
- Functional Hardware
- DTP

Research Question

Question

Method

DTP / Agda

Big picture
Agda

Π-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 : $\mathbb{C}' \text{ zero zero}$

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

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

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

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

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -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 : $\forall i \rightarrow \text{atom} \rightarrow n \ (n \rightarrow \text{atom} \ i) \equiv i$

inv-right : $\forall a \rightarrow n \rightarrow \text{atom} \ (\text{atom} \rightarrow n \ a) \equiv a$

|Atom| = suc |Atom|−1

Atom# = Fin |Atom|

Background

Hardware Design

Functional Hardware

DTP

Research

Question

Question

Method

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

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

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

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

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



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**

Background

- Hardware Design
- Functional Hardware
- DTP

Research Question

Question

Method

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|

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



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)))
```

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



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  ]
```

$$\text{spec-not} \quad (x :: \varepsilon) \quad = \text{[not } x \text{]}$$

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

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

Background

Hardware Design

Functional Hardware

DTP

Research

Question

Question

Method

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 }
```

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



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 α and β
 - Passed as *instance arguments*

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π-Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



Synthesizable

- ▶ $\Downarrow W \Uparrow$ type class (pronounced Synthesizable)
 - Describes how to *synthesize* a given Agda type (α)
 - Two fields: from element of α to a *word* and back

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

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

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -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 (`_×_`)

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

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

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

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

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

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

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



Synthesizable

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

Background

- Hardware Design
- Functional Hardware
- DTP

Research Question

- Question
- Method

DTP / Agda

- Big picture
- Agda

Π -Ware

- Syntax**
- Semantics
- Proofs

Conclusions

- Limitations
- Future work



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

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



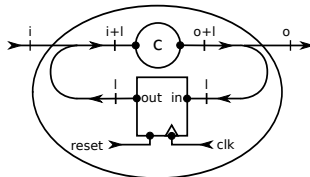
Synthesis

- Netlist: directed graph where nodes are *gates* and edges are *buses*
- Work-in-progress

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

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

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

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


Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π-Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



Universiteit Utrecht

Simulation

- ▶ Combinational
- ▶ Sequential

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



Examples

► AndN

Background

- Hardware Design
- Functional Hardware
- DTP

Research Question

- Question
- Method

DTP / Agda

- Big picture
- Agda

Π -Ware

- Syntax
- Semantics

Proofs

Conclusions

- Limitations
- Future work



Problems

- Definition of $\llbracket _ \rrbracket$ blocks reduction

Background

Hardware Design

Functional Hardware

DTP

Research Question

Question

Method

DTP / Agda

Big picture

Agda

Π-Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

Summary

► Π-Ware is...

Background

- Hardware Design
- Functional Hardware
- DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π-Ware

- Syntax
- Semantics
- Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht

Current limitations

- ▶ Problem with proofs (definition of $\llbracket_ \rrbracket$)
- ▶ Proofs on (infinite) **Streams**
- ▶ Bla

Background

Hardware Design
Functional Hardware
DTP

Research Question

Question
Method

DTP / Agda

Big picture
Agda

Π -Ware

Syntax
Semantics
Proofs

Conclusions

Limitations
Future work



Future work

- Proof by reflection for finite cases

Background

- Hardware Design
- Functional Hardware
- DTP

Research Question

Question
Method

DTP / Agda

- Big picture
- Agda

Π-Ware

- Syntax
- Semantics
- Proofs

Conclusions

Limitations

Future work



Thank you!

Questions?



References I



Lava: hardware design in Haskell.

Sheeran, M. (1984).

In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 104–112, New York, NY, USA. ACM.



Propositions as types.

Unpublished note, <http://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf>.

Future work

