

# $\Pi$ -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

Sunday 24<sup>th</sup> August, 2014

Background

Hardware Design  
Functional Hardware  
DTP

Research  
Question

Question  
Method

DTP / Agda

Big picture  
Agda

$\Pi$ -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

## $\Pi$ -Ware

Syntax

Semantics

Proofs

## Conclusions

Limitations

Future work

## Background

Hardware Design

Functional Hardware

DTP

## Research Question

Question

Method

## DTP / Agda

Big picture

Agda

## $\Pi$ -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

## $\Pi$ -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

## $\Pi$ -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

## $\Pi$ -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

## $\Pi$ -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,  $\mu$ 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

## $\Pi$ -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

## $\Pi$ -Ware

- Syntax
- Semantics
- Proofs

## Conclusions

- Limitations
- Future work



# Methodology

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

## Background

Hardware Design  
Functional Hardware  
DTP

## Research Question

Question  
Method

## DTP / Agda

Big picture  
Agda

## $\Pi$ -Ware

Syntax  
Semantics  
Proofs

## Conclusions

Limitations  
Future work



# Dependently-Typed Programming

## ► Types can depend on values

- Example: `data Vec ( $\alpha$  : Set) :  $\mathbb{N} \rightarrow$  Set where...`
- Compare with Haskell (GADT style):  
`data List :: * -> * where...`

## ► Types of arguments can depend on *values of previous arguments*

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

### Background

Hardware Design  
Functional Hardware  
DTP

### Research Question

Question  
Method

### DTP / Agda

Big picture  
Agda

### $\Pi$ -Ware

Syntax  
Semantics  
Proofs

### Conclusions

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 ]`

## Background

Hardware Design  
Functional Hardware  
DTP

## Research Question

Question  
Method

## 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

## Background

- Hardware Design
- Functional Hardware
- DTP

## Research Question

- Question
- Method

## DTP / Agda

- Big picture
- Agda

## $\Pi$ -Ware

- Syntax
- Semantics
- Proofs

## Conclusions

- Limitations
- Future work



# Dependently-Typed Programming

## ► Dependent pattern matching can *rule out* impossible cases

- Classic example: *safe head* function

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

$\text{head } (x :: xs) = x$

### Background

Hardware Design  
Functional Hardware  
DTP

### Research Question

Question  
Method

### DTP / Agda

Big picture  
Agda

### $\Pi$ -Ware

Syntax  
Semantics  
Proofs

### Conclusions

Limitations  
Future work



# 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  $\Pi$ -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`

## Background

Hardware Design  
Functional Hardware  
DTP

## Research Question

Question  
Method

## 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:

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

## 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

## $\Pi$ -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 : Set<sub>1</sub> 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
$$|B|-1 = 1$$
$$|B| = \text{succ } |B|-1$$
- ▶ Friendlier names for the indices (elements of **Fin 2**)
$$\text{pattern False\#} = Fz$$
$$\text{pattern True\#} = Fs Fz$$
$$\text{pattern Absurd\# } n = Fs (Fs n)$$

## Background

Hardware Design  
Functional Hardware  
DTP

## Research Question

Question  
Method

## DTP / Agda

Big picture  
Agda

## $\Pi$ -Ware

Syntax  
Semantics  
Proofs

## Conclusions

Limitations  
Future work



# Atomic instance (Bool)

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

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

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

$$\text{inv-left-B} = \lambda \{ \text{False\#} \rightarrow \text{refl}; \text{True\#} \rightarrow \text{refl}; (\text{Absurd\# } ()) \}$$
$$\text{inv-right-B} = \lambda \{ \text{false} \rightarrow \text{refl}; \text{true} \rightarrow \text{refl} \}$$

- 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



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](#)

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

Gates# = Fin |Gates|

## Background

Hardware Design  
Functional Hardware  
DTP

## Research Question

Question  
Method

## DTP / Agda

Big picture  
Agda

## $\Pi$ -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)))
pattern Absurd# n    = Fs (Fs (Fs (Fs (Fs n))))
```

## 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



► “Typed”

## 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

# Synthesizable

►  $\Downarrow W \Uparrow$  (pronounced Synthesizable)

- $W\ n = \text{Vec}\ \alpha\ n$

► Example:  $\Downarrow W \Uparrow\ (\alpha \times \beta)$

## Background

Hardware Design  
Functional Hardware  
DTP

## Research Question

Question  
Method

## DTP / Agda

Big picture  
Agda

## $\Pi$ -Ware

Syntax  
Semantics  
Proofs

## Conclusions

Limitations  
Future work



# Synthesis

- ▶ Work-in-progress
- ▶ **Atom** and **Gates** with VHDL *abstract syntax*

## Background

Hardware Design  
Functional Hardware  
DTP

## Research Question

Question  
Method

## DTP / Agda

Big picture  
Agda

## $\Pi$ -Ware

Syntax  
**Semantics**  
Proofs

## Conclusions

Limitations  
Future work



# Simulation

- ▶ Combinational
- ▶ Sequential

## Background

Hardware Design  
Functional Hardware  
DTP

## Research Question

Question  
Method

## DTP / Agda

Big picture  
Agda

## $\Pi$ -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



Universiteit Utrecht

# 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

## $\Pi$ -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

## $\Pi$ -Ware

- Syntax
- Semantics
- Proofs

## Conclusions

- Limitations
- Future work**



Thank you!

Questions?



# References I

 Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. (1998).

Lava: hardware design in Haskell.

*SIGPLAN Not.*, 34(1):174–184.

 Sheeran, M. (1984).

MuFP, a language for VLSI design.

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

 Wadler, P. (2014).

Propositions as types.

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

Background

Hardware Design

Functional Hardware

DTP

Research

Question

Question

Method

DTP / Agda

Big picture

Agda

$\Pi$ -Ware

Syntax

Semantics

Proofs

Conclusions

Limitations

Future work



Universiteit Utrecht