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

Universiteit Utrecht

Table of Contents

Hardware design is hard(er)

- Strict(er) correctness requirements
  - You can't simply *update* a full-custom chip after production
    - Intel `FDIV`
  - Expensive verification / validation (up to 50% of development costs)
- Low-level details (more) important
  - Layout / area
  - Power consumption / fault tolerance

Hardware design is growing

- Moore's law will still apply for some time
  - We can keep packing more transistors into same silicon area
- **But** optimizations in CPUs display diminishing returns
  - Thus, more algorithms *directly* in hardware

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

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

Embedded DSLs for Hardware

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

Dependently-Typed Programming

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

Universiteit Utrecht

Research Question

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

Methodology

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

Dependently-Typed Programming

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

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:

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

    - This does not:

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

Dependently-Typed Programming

- Dependent pattern matching can *rule out* impossible cases

Dependently-Typed Programming

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

$$\mathsf{head} : \mathsf{Vec}\ \alpha\ (\mathsf{suc}\ n) \to \alpha$$
$$\mathsf{head}\ (x :: xs) = x$$

Dependently-Typed Programming

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

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

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

Depedent 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

Agda syntax for Haskell programmers

- Liberal identifier lexing (Unicode **everywhere**)
  - a≡b+c is a valid identifer, $a \equiv b + c$ an expression
  - Actually used in Agda's standard library
  - And in Π-Ware: ℂ, ⟦ c ⟧, ⇓, ⇑
- *Mixfix* notation
  - _[_]≔_ is the vector update function: v [# 3] ≔ true.
  - _[_]≔_ v (# 3) true ⟺ v [# 3] ≔ true
- Almost nothing built-in
  - _+_ : ℕ → ℕ → ℕ defined in Data.Nat
  - if_then_else_ : Bool → $\alpha \to \alpha \to \alpha$ defined in Data.Bool

Agda syntax for Haskell programmers

- Implicit arguments
  - Don't have to be passed if Agda can **guess** it
  - Syntax: $\varepsilon : \{\alpha : \mathsf{Set}\} \rightarrow \mathsf{Vec}\ \alpha\ \mathsf{zero}$
- "For all" syntax: $\forall\ n \iff (n : \_)$
  - Where _ means: guess this type (based on other args)
  - Example:
    - $\forall\ n \rightarrow \mathsf{zero} \leq n$
    - $\mathsf{data}\ \_\leq\_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathsf{Set}$
- It's common to combine both:
  - $\forall\ \{\alpha\ n\} \rightarrow \mathsf{Vec}\ \alpha\ (\mathsf{suc}\ n) \rightarrow \alpha \iff$
    $\{\alpha : \_\}\ \{n : \_\} \rightarrow \mathsf{Vec}\ \alpha\ n \rightarrow \alpha$

Low-level circuits

- Structural representation
- Untyped but *sized*

```
data ℂ' : ℕ → ℕ → Set
data ℂ' where
    Nil      : ℂ' zero zero
    Gate     : (g# : Gates#)  → ℂ' (|in| g#) (|out| g#)
    Plug     : ∀ {i o}        → (f : Fin o → Fin i) → ℂ' i o

    DelayLoop : (c : ℂ' (i + l) (o + l)) {comb' c} → ℂ' i o

    _⟫'_   : ℂ' i m    → ℂ' m o    → ℂ' i o
    _|'_   : ℂ' i₁ o₁  → ℂ' i₂ o₂  → ℂ' (i₁ + i₂) (o₁ + o₂)
    _|+'_  : ℂ' i₁ o   → ℂ' i₂ o   → ℂ' (suc (i₁ ⊔ i₂)) o
```

Atoms

- How to carry values of an Agda type in *one* wire
- Defined by the Atomic type class in PiWare.Atom

```
record Atomic : Set₁ where
  field
    Atom       : Set
    |Atom|-1   : ℕ
    n→atom     : Fin (suc |Atom|-1) → Atom
    atom→n     : Atom → Fin (suc |Atom|-1)

    inv-left   : ∀ i → atom→n (n→atom i) ≡ i
    inv-right  : ∀ a → n→atom (atom→n a) ≡ a

  |Atom| = suc |Atom|-1
  Atom#  = Fin |Atom|
```

Atomic instances

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

```
|B|-1 = 1
|B| = suc |B|-1
```

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

```
pattern False#  = Fz
pattern True#   = Fs Fz
```


Universiteit Utrecht

Atomic instance (Bool)

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

```
n→B = λ { False# → false;   True#  → true }
B→n = λ { false  → False#;  true   → True# }
```

- Proof that n→B and B→n are inverses

```
inv-left-B   = λ { False# → refl;  True# → refl; }
inv-right-B  = λ { false  → refl;  true  → refl }
```

- With all pieces at hand, we construct the instance

```
Atomic-B = record { Atom      = B
                  ; |Atom|-1  = |B|-1
                  ; n→atom    = n→B
                  ; atom→n    = B→n
                  ; inv-left  = inv-left-B
                  ; inv-right = inv-right-B }
```

Gates

- Circuits parameterized by collection of *fundamental gates*
- Examples:
  - {NOT, AND, OR} (BoolTrio)
  - {NAND}
  - Arithmetic, Crypto, etc.
- The definition of what means to be such a collection is in PiWare.Gates.Gates

The Gates type class

```
W : ℕ → Set
W = Vec Atom
 record Gates : Set where

   field
      |Gates|     : ℕ
      |in| |out|  : Fin |Gates| → ℕ
      spec        : (g : Fin |Gates|)
                    → (W (|in| g) → W (|out| g))

   Gates#  = Fin |Gates|
```

Gates instances

- Example: PiWare.Gates.BoolTrio
- First, how many gates are there in the library

  $|\text{BoolTrio}| = 5$

- Then the friendlier names for the indices

```
pattern FalseConst#  = Fz
pattern TrueConst#   = Fs Fz
pattern Not#         = Fs (Fs Fz)
pattern And#         = Fs (Fs (Fs Fz))
pattern Or#          = Fs (Fs (Fs (Fs Fz)))
```

Gates instance (BoolTrio)

- Defining the *interfaces* of the gates

$$
\begin{aligned}
|\mathsf{in}|\ \mathsf{FalseConst\#} &= 0\\
|\mathsf{in}|\ \mathsf{TrueConst\#} &= 0\\
|\mathsf{in}|\ \mathsf{Not\#} &= 1\\
|\mathsf{in}|\ \mathsf{And\#} &= 2\\
|\mathsf{in}|\ \mathsf{Or\#} &= 2
\end{aligned}
$$

$$|\mathsf{out}|\ \_ = 1$$

- And the specification function for each gate

$$
\begin{aligned}
\mathsf{spec{-}false}\ &\_ &&= [\ \mathsf{false}\ ]\\
\mathsf{spec{-}true}\ &\_ &&= [\ \mathsf{true}\ ]\\
\mathsf{spec{-}not}\ &(x :: \varepsilon) &&= [\ \mathsf{not}\ x\ ]\\
\mathsf{spec{-}and}\ &(x :: y :: \varepsilon) &&= [\ x \wedge y\ ]\\
\mathsf{spec{-}or}\ &(x :: y :: \varepsilon) &&= [\ x \vee y\ ]
\end{aligned}
$$

Gates instance (BoolTrio)

- Mapping each gate index to its respective specification

```
specs-BoolTrio FalseConst#  = spec-false
specs-BoolTrio TrueConst#   = spec-true
specs-BoolTrio Not#         = spec-not
specs-BoolTrio And#         = spec-and
specs-BoolTrio Or#          = spec-or
```

- With all pieces at hand, we construct the instance

```
BoolTrio : Gates
BoolTrio = record { |Gates| = |BoolTrio|
                  ; |in|    = |in|
                  ; |out|   = |out|
                  ; spec    = specs-BoolTrio }
```

High-level circuits

- $\mathbb{C}$
- "Typed"

Synthesizable

- $\Downarrow$W$\Uparrow$ (pronouced `Synthesizable`)
  - W n = Vec α n
- Example: $\Downarrow$W$\Uparrow$ $(\alpha \times \beta)$

Synthesis

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

Simulation

- Combinational
- Sequential

Examples

- `AndN`

Problems

- Definition of $[\![\_]\!]$ blocks reduction

Universiteit Utrecht

Summary

- Π-Ware is...

Current limitations

- Problem with proofs (definition of $[\![\_]\!]$)
- Proofs on (infinite) Streams
- Bla

Future work

- Proof by reflection for finite cases

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`.