Π-Ware: Hardware Description with Dependent Types

Author: João Paulo Pizani Flor
<j.p.pizani@uu.nl>

Supervisor: Wouter Swierstra
<w.s.swierstra@uu.nl>

Department of Information and Computing Sciences
Utrecht University

Monday 18th May, 2015

Universiteit Utrecht

Context

Universiteit Utrecht

One-sentence definition

A unified DSL (Π-Ware) embedded in *Agda* for *modeling* hardware circuits, *synthesizing* them and *proving* properties about their behaviour and structure.

Universiteit Utrecht

Hardware is growing

More specifically, hardware *acceleration*. Three reasons why:

- Miniaturization still has some generations to go [3]
- Microarch. optimization gives diminishing returns [1]
- Battery energy density vs. demand for computation

More applications benefit from *hardware acceleration*

- DSP, crypto, codecs, graphics, comm. protocols, etc.

Hardware design benefits more from *rigour*

- *Early optimization*, more error-prone
- Mass production, less *updateable*

Hardware design "status quo"

Myriad of languages for specific design tasks...

- **Simulation:** SystemC, VHDL/Verilog
- **Synthesis:** VHDL/Verilog (subsets), C/C++ (subsets)
- **Verification:** SAT solvers / Theorem provers

Problems:

- Manual translation
- Loss of invariants, manual checking

An analogous situation in software seems bizarre:

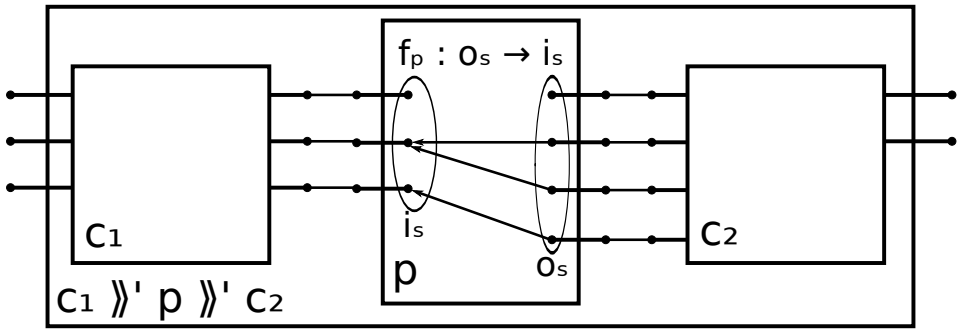
- To "simulate" (interpret) your program, you use Haskell
- For compilation to x86, use C (non-standardized)

Functional hardware DSLs

- Solve **most** of the problems with multiple descriptions
- "Popular" example: Lava (Chalmers)
  - Description, simulation, testing in Haskell
  - Verification through external SAT solver
- Drawbacks:
  - Non modular verification (fully-automated)
  - Only for *specific* circuits (not *families*)
  - Haskell types not expressive enough
    - `addN :: Int -> ([Bit], [Bit]) -> [Bit]`
    - Could use lots of extensions, but why compromise?

Dependent types for hardware

- Well-formedness
- Rule out design mistakes *early*
  - Floating wires (matching interfaces)
  - Short-circuits (Plug constructor)



- More precise specification of circuit *generators*
  - **Haskell:** `addN :: Int -> ([Bit], [Bit]) -> [Bit]`
  - **Agda:** addN : $(n : \mathbb{N}) \to$ C $(2 * n)$ $(\text{suc}\ n)$
- Mainly: proofs in the same language as the models
  - (Functional) correctness proofs
  - Provably-correct circuit *transformations*

Universiteit Utrecht

Π-Ware

Universiteit Utrecht

Circuit syntax

- Π-Ware is *deep-embedded*
  - Multiple semantics, algebraic manipulation
- Low-level, *architectural* representation
  - Analogous to a block diagram
  - Untyped, but *sized*

$$
\begin{aligned}
&\textbf{data}\ \mathbb{C}\ \textbf{where}\\
&\quad \mathsf{Gate} \;:\; \mathbb{C}\ (|\mathsf{in}|\ g)\ (|\mathsf{out}|\ g)\\
&\quad \mathsf{Plug} \;:\; i \leftrightarrows o \to \mathbb{C}\ i\ o\\
&\quad \_\gg\_ \;:\; \mathbb{C}\ i\ m \to \mathbb{C}\ m\ o \to \mathbb{C}\ i\ o\\
&\quad \_\|\_ \;:\; \mathbb{C}\ i_1\ o_1 \to \mathbb{C}\ i_2\ o_2 \to \mathbb{C}\ (i_1 + i_2)\ (o_1 + o_2)
\end{aligned}
$$

Universiteit Utrecht

Circuit syntax

- Combinational / sequential
  - Single way of constructing a sequential circuit: DelayLoop

$$\mathsf{DelayLoop} : \mathbb{C}\ \{\sigma\}\ (i + l)\ (o + l) \to \mathbb{C}\ \{\omega\}\ i\ o$$

- The $\mathbb{C}$ type is "tagged" to keep the two cases distinct
  - The distinction is mainly important for simulation
  - Easier definitions of *generators*
    - ```
      data ℂ : {p : IsComb} → Ix → Ix → Set
      data IsComb : Set where
        σ ω : IsComb
      ```
  - Obs: ω has to do with Σ^ω


      ```

Fundamental gates

- Circuits are built by combining smaller circuits
  - Ultimately, from a library of *fundamental* Gates
  - Each gate specified by a *function* over (binary) *words*

```
andSpec : Vec Bool 2 → Vec Bool 1
andSpec (x :: y :: ε) = [ x ∧ y ]
```

- A "traditional" instance of Gates is BoolTrio
  - Set of gates: $\{\bot, \top, \neg, \wedge, \vee\}$
  - With the usual specification (stdlib)
- Other "interesting" instances:
  - Modular arithmetic
  - Cryptographic primitives
  - Primitives for *scans* (case study)

Fundamental gates

- To define a gate library, we need to define:
  - How many gates are there
  - Each gate's *interface*
  - Each gate's *specification*

$$\begin{aligned} |\mathsf{in}|\ |\mathsf{out}| &: \mathsf{Gate\#} \to \mathbb{N} \\ \mathsf{spec} &: (g : \mathsf{Gate\#}) \to (\mathsf{W}\ (|in|\ g) \to \mathsf{W}\ (|out|\ g)) \end{aligned}$$

- Dependent types help us again
  - The Gate# type ranges in $[0..n-1]$
  - spec works over words of the *right size*

Universiteit Utrecht

Atomic types

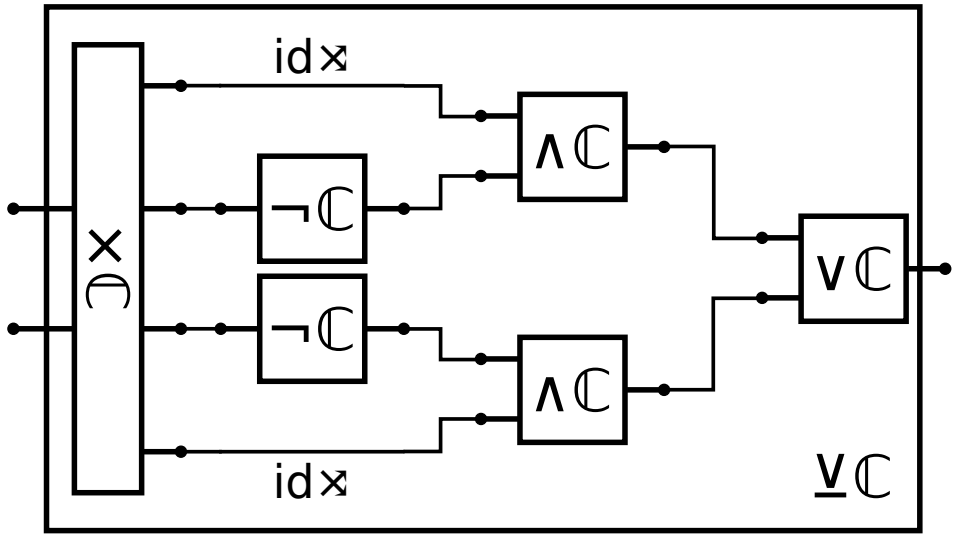
- The whole Circuit module is parameterized by a record
  - Defining what is *carried* over the "wires"
  - W = Vec Atom
- This Atomic class is similar to Haskell's Enum
  - An atomic type needs to be *finite*
  - There's a bijection between the type and $[0..n-1]$
    - enum : Fin $|Atom| \leftrightarrow Atom$
    - In Agda, the bijection is *proven*
- Dependent types move runtime errors to type checking:
  - **Haskell**: `succ maxBound` $\rightarrow$ runtime error
  - **Agda**: "`succ maxBound`" $\rightarrow$ doesn't typecheck!

Atomic types (Bool)

- Some possible instances...
  - Bool
  - Multi-valued logics (VHDL's `std_logic`)
  - States of a state machine
- Simplest "useful": Bool
  - We use the mapping $0 \leftrightarrow False; 1 \leftrightarrow True$
  - Order and choice of indices *don't matter*
- Later how this parameterization ties into *synthesis*

Putting all pieces together

- Small circuit using Bool atoms and BoolTrio gates



$$\underline{\vee}\mathbb{C} : \mathbb{C}\ 2\ 1$$
$$\underline{\vee}\mathbb{C} = \mathsf{fork}\!\!\nwarrow\!\!\nearrow \gg (\neg\mathbb{C} \parallel \mathsf{id}\!\!\nwarrow\!\!\nearrow \gg \wedge\mathbb{C}) \parallel (\mathsf{id}\!\!\nwarrow\!\!\nearrow \parallel \neg\mathbb{C} \gg \wedge\mathbb{C}) \gg \vee\mathbb{C}$$

Universiteit Utrecht

Data abstraction

- Sometimes it's more convenient to have *typed* circuit I/O
  - Used for conveniently-typed testing/simulation
  - $\mathbb{C}$ (Bool × Bool) Bool instead of $\mathbb{C}$ 2 1
- To be used as circuit I/O, a type needs to be Synthesizable
  - Have a mapping to vectors of Atoms (a.k.a *words*)

```
record ⇓W⇑ (α : Set) {i : Ix} : Set where
  constructor ⇓W⇑[_,_]
  field  ⇓ : α → W i
         ⇑ : W i → α
```

- Instances
  - Currently: _×_, _⊎_, Vec, primitives.
  - Future: datatype-generic approach

Universiteit Utrecht

Circuit semantics

- Our goal is to have two semantics:
  - Behavioural (done)
  - Structural/Synthesis (TODO)
- Our behavioural semantics is *functional*
  - From a circuit, a *function* is derived
  - Circuits can be "run" or simulated over inputs

Circuit semantics

- Our goal is to have two semantics:
  - Behavioural (done)
  - Structural/Synthesis (TODO)
- Our behavioural semantics is *functional*
  - From a circuit, a *function* is derived
  - Circuits can be "run" or simulated over inputs
- Two kinds of simulation: *combinational* and *sequential*
  - **Combinational:** no internal state
    - $[\![\_]\!] : \mathbb{C}\ \{\sigma\}\ i\ o \to (\mathsf{W}\ i \to \mathsf{W}\ o)$
      $[\![\_]\!] = \mathsf{cata}\mathbb{C}\sigma\ \{Al = \mathsf{simulation{-}combinational}\}$
  - *Tagged* with σ (has no DelayLoop)
  - Example: $[\![\ \mathsf{and}\ ]\!]\ (\mathsf{true} :: \mathsf{false} :: \varepsilon) \equiv [\ \mathsf{false}\ ]$

Sequential simulation

- More general, for circuit with (possibly) internal state
  - Simulation works over infinite sequences
  - Modeled using Agda's Stream (coinductive)
- User interface
  - $[\![\_]\!]\omega : \mathbb{C}\ i\ o \to (\mathsf{Stream}\ (\mathsf{W}\ i) \to \mathsf{Stream}\ (\mathsf{W}\ o))$
    $[\![\_]\!]\omega = \mathsf{run} \circ [\![\_]\!]*$
  - Works on **both** sequential and combinational circuits
  - Ex: $[\![\ \mathsf{not}\ ]\!]\omega\ (\mathsf{repeat}\ [\ \mathsf{false}\ ]) \approx \mathsf{repeat}\ [\ \mathsf{true}\ ]$
- Stream functions can "look into the future"
  - We use a *causal stream function*
  - $f = \mathsf{run} \circ f'$ with f' a *step* (past $\times$ present $\to$ next)
  - Idea from Tarmo Uustalu's paper [4]

Proving circuit properties

- What can be proven: depends on which semantics is used
  - **Structural**: "the circuit size grows linearly with input size"
  - **Behavioural:** "the circuit will never produce value X"
- Example behavioural property: *functional correctness*
  - Agreement with a *specification* on all inputs
  - Specification is a *function*
  - Ex: $\forall\ (x\ y : \mathsf{Int8}) \to [\![\ \mathsf{add}_{256}\ ]\!]\ (x\ ,\ y) \equiv x +_{256} y$

Universiteit Utrecht

Properties of circuit combinators

- Circuit combinators have *algebraic* properties
  - $\{\ \_\gg\_,\ \mathsf{id}\!\!\times\ \}$ forms an (indexed) monoid
  - $\{\ \_\|\_,\ \mathsf{nil}\!\!\times\ \}$ forms an (indexed) monoid
  - $f \circ g \cong \mathsf{id} \to \mathsf{Plug}\ g \gg \mathsf{Plug}\ f \approx \mathsf{id}\!\!\times$
- Developed a notion of equivalence between circuits ($\_\approx\_$)
  - Equality *up to a certain semantics* (simulation)
    - Proven equivalence relation
    - Equational reasoning
  - Combinators are *congruences* ($\_\gg\_$, $\_\|\_$, Plug, *custom*)
  - Equal behaviour $\longrightarrow$ opportunity for *optimization*
- Π-Ware can be used to define whole *classes* of circuits
  - With their own associated laws...

Universiteit Utrecht

Present / Future

Universiteit Utrecht

Current work

Finishing up...

- Case study: parallel-prefix circuits
  - Computes $[a_1, (a_1 + a_2), (a_1 + a_2 + a_3), ...]$ in parallel
  - Behaviour similar to Haskell's `scanl`
  - Applications: sorting, addition, filters, etc., etc.
- General class + examples implemented in Π-Ware
  - Inspired by Ralf Hinze's "An algebra of scans" [2]
  - As M.Sc experimentation project (Yorick Sijsling)
  - Ex. law: scan (suc m) ▱ scan (suc n) ≋ scan (m + suc n)
- "Side-effects" of the project

Universiteit Utrecht

Current work

Partly there / beginning...

- Correctness of sequential circuits
  - Temporal logic
  - Difficulties with *bisimilarity* (_≈_)
    - Better *carrier* than (Stream α → Stream β)?
- Little testing framework
- "Automated" checking for any *specific* circuit
  - Given a (trivial) proofs, one for each possible input
  - Produce proof of generalized statement
  - Already working for Fin n
    - elimFin : Vec↑I P n (allFin n) → ($\forall$ i → P i)

Future

- Translation to VHDL
  - Simplified, intermediary language
  - Two *key* additions to the framework
  - In Atomic: VHDL type, one VHDL expression per value
  - In Gates: one VHDL component per gate
- Optimizations in generated VHDL
  - Try to use circuit laws to justify "rewrite" steps
  - Example: $((a_1 \wedge a_2) \wedge a_3) \wedge a_4 \cong (a_1 \wedge a_2) \wedge (a_3 \wedge a_4)$
- Automation possibilities
  - Testing input generation
  - Congruence "generation"
  - Monoid solver for circuit equality

Thank you!

Questions?

`https://github.com/joaopizani/piware-agda`
`https://github.com/yoricksijsling/PiWare-prefixes`

Universiteit Utrecht

References I

H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger.
Dark silicon and the end of multicore scaling.
In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

Ralf Hinze.
An algebra of scans.
In Dexter Kozen, editor, *Mathematics of Program Construction*, number 3125 in Lecture Notes in Computer Science, pages 186–210. Springer Berlin Heidelberg, January 2004.

Universiteit Utrecht

References II

Bernd Hoefflinger.
ITRS: The international technology roadmap for semiconductors.
In Bernd Hoefflinger, editor, *Chips 2020*, The Frontiers Collection, pages 161–174. Springer Berlin Heidelberg, January 2012.

Tarmo Uustalu and Varmo Vene.
The essence of dataflow programming.
In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 2–18, Berlin, Heidelberg, 2005. Springer-Verlag.

Universiteit Utrecht