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
Context
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.
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
    - \( \text{addN} :: \text{Int} \rightarrow ([\text{Bit}], [\text{Bit}]) \rightarrow [\text{Bit}] \)
    - 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)

![Diagram of circuit components]

► More precise specification of circuit *generators*
  • **Haskell:** `addN :: Int -> ([Bit], [Bit]) -> [Bit]`
  • **Agda:** `addN : (n : ℕ) → C (2 * n) (suc n)`

► Mainly: proofs in the same language as the models
  • (Functional) correctness proofs
  • Provably-correct circuit *transformations*
Π-Ware
Circuit syntax

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

```haskell
data C where
  Gate : C (|lin| g) (|out| g)
  Plug : i ⊗ o → C i o
  _⟩⟩_ : C i m → C m o → C i o
  _∥∥_ : C i₁ o₁ → C i₂ o₂ → C (i₁ + i₂) (o₁ + o₂)
```
Circuit syntax

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

\[
\text{DelayLoop} : \mathbb{C} \{\sigma\} (i + l) (o + l) \rightarrow \mathbb{C} \{\omega\} i o
\]

- The \(\mathbb{C}\) type is “tagged” to keep the two cases distinct
  - The distinction is mainly important for simulation
  - Easier definitions of generators
    - data \(\mathbb{C} : \{p : \text{IsComb}\} \rightarrow \text{Ix} \rightarrow \text{Ix} \rightarrow \text{Set}\)
      - data IsComb : Set where \(\sigma \omega : \text{IsComb}\)
    - Obs: \(\omega\) has to do with \(\Sigma^\omega\)
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

\[
\text{andSpec} : \text{Vec Bool} \ 2 \rightarrow \text{Vec Bool} \ 1 \\
\text{andSpec} (x :: y :: \varepsilon) = [x \land y]
\]

- A “traditional” instance of Gates is BoolTrio
  - Set of gates: \{\bot, \top, \neg, \land, \lor\}
  - With the usual specification (stdlib)

- Other “interesting” instances:
  - Modular arithmetic
  - Cryptographic primitives
  - Primitives for scans (case study)
To define a gate library, we need to define:

- How many gates are there
- Each gate’s *interface*
- Each gate’s *specification*

\[
|\text{in}| \ |\text{out}| \ : \ \text{Gate#} \rightarrow \mathbb{N}
\]

\[
\text{spec} \ : \ (g \ : \ \text{Gate#}) \rightarrow (W \ (|\text{in}| \ g) \rightarrow W \ (|\text{out}| \ g))
\]

Dependent types help us again

- The *Gate#* type ranges in [0..\(n - 1\)]
- *spec* works over words of the *right size*
Atomic types

- The whole Circuit module is parameterized by a record
  - Defining what is carried over the “wires”
  - $W = \text{Vec} \ Atom$

- This Atomic class is similar to Haskell’s Enum
  - An atomic type needs to be finite
  - There’s a bijection between the type and $[0..n - 1]$
    - $\text{enum} : \text{Fin} \ |Atom| \leftrightarrow Atom$
    - In Agda, the bijection is proven

- Dependent types move runtime errors to type checking:
  - Haskell: $\text{succ} \ \text{maxBound} \rightarrow$ runtime error
  - Agda: “$\text{succ} \ \text{maxBound}$” $\rightarrow$ 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 influences *synthesis*
Putting all pieces together

- Small circuit using **Bool** atoms and **BoolTrio** gates

\[ ∀C : C \ 2 \ 1 \]
\[ ∀C = \text{fork} \times \]
\[ \equiv (\neg C \ || \ \text{id} \times \ ) \ \land C \) \ || \ (\text{id} \times \ || \ \neg C \ ) \ \land C \]
\[ \equiv ∀C \]
Data abstraction

▶ Sometimes it’s more convenient to have typed circuit I/O
  - Used for conveniently-typed testing/simulation
  - \( C (\text{Bool} \times \text{Bool}) \text{ Bool} \) instead of \( C \ 2 \ 1 \)

▶ 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 \( \downarrow W \uparrow (\alpha : \text{Set}) \{ i : \text{Ix} \} : \text{Set} \) where
  constructor \( \downarrow W \uparrow[\_, \_\] \)
  field \( \downarrow : \alpha \rightarrow W \ i \)
  \( \uparrow : W \ i \rightarrow \alpha \)

▶ Instances
  - Currently: \( \_\times\_, \_\uplus\_, \text{Vec}, \) primitives.
  - Future: datatype-generic approach
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
    - $⟦_⟧ : \mathbb{C} \{\sigma\} \ i \ o \to (\mathbb{W} \ i \to \mathbb{W} \ o)$
    - $⟦_⟧ = \text{cataC} \sigma \ \{Al = \text{simulation–combinational}\}$
  - *Tagged* with $\sigma$ (has no DelayLoop)
  - Example: $⟦\ \text{and} \ ⟧ (\text{true} :: \text{false} :: \varepsilon) \equiv [\text{false}]$
Sequential simulation

- More general, for circuit with (possibly) internal state
  - Simulation works over infinite sequences
  - Modeled using Agda’s Stream (coinductive)

- User interface
  - $\omega : C \rightarrow (\text{Stream } W i \rightarrow \text{Stream } W o)$
  - $\omega = \text{run} \circ [\_]*$
  - Works on both sequential and combinational circuits
  - Ex: $[\text{not}]\omega (\text{repeat} [\text{false}]) \approx \text{repeat} [\text{true}]

- Stream functions can “look into the future”
  - We use a causal stream function
  - $f = \text{run} \circ f^\prime$ with $f^\prime$ a step (past $\times$ present $\rightarrow$ 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: \( \forall (x \ y : \text{Int8}) \rightarrow \llbracket \text{add}_{256} \rrbracket (x , y) \equiv x +_{256} y \)
Properties of circuit combinators

- Circuit combinators have \textit{algebraic} properties
  - \{ \_⟫\_, \_∥\_ \} forms an (indexed) monoid
  - \{ \_\parallel\_, \_\parallel\_ \} forms an (indexed) monoid
  - \( f \circ g \cong \text{id} \rightarrow \text{Plug } g \circ \text{Plug } f \cong \text{id} \)

- Developed a notion of equivalence between circuits (\_\cong\_)
  - Equality \textit{up to a certain semantics} (simulation)
    - Proven equivalence relation
    - Equational reasoning
  - Combinators are \textit{congruences} (\_⟫\_, \_∥\_, \text{Plug}, \text{custom})
  - Equal behaviour \(\rightarrow\) opportunity for \textit{optimization}

- \(\Pi\)-Ware can be used to define whole \textit{classes} of circuits
  - With their own associated laws...
Present / Future
Current work

Finishing up…

- Case study: parallel-prefix circuits
  - Computes \([a_1, (a_1 + a_2), (a_1 + a_2 + a_3), \ldots]\) in parallel
  - Behaviour similar to Haskell’s \texttt{scan1}
  - 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: \(\text{scan} \ (\text{suc} \ m) \cong \text{scan} \ (\text{suc} \ n) \approxeq \text{scan} \ (m + \text{suc} \ n)\)

- “Side-effects” of the project
Current work

Partly there / beginning...

- Correctness of sequential circuits
  - Temporal logic
  - Difficulties with \textit{bisimilarity} (\_\_\_\_\_\_)
    - Better \textit{carrier} than (\text{Stream } \alpha \to \text{Stream } \beta)\?

- Little testing framework

- “Automated” checking for any \textit{specific} circuit
  - Given a (trivial) proofs, one for each possible input
  - Produce proof of generalized statement
  - Already working for \text{Fin } n
    - \texttt{elimFin} : \text{Vec}↑Ι P n (\text{allFin } n) \to (\forall i \to 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: \(((a_1 \land a_2) \land a_3) \land a_4 \cong (a_1 \land a_2) \land (a_3 \land a_4)\)

▶ 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
Dark silicon and the end of multicore scaling.

Ralf Hinze.
An algebra of scans.
References II
