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

Friday 9th January, 2015
Definition
A unified DSL (Π-Ware) embedded in Agda for *modelling* hardware circuits, *synthesizing* them and *proving* properties about their behaviour and structure.
Motivation
Hardware is growing

More specifically, hardware *acceleration*

▶ Miniaturization still has a decade to go [3]
  - Microarch. optimization shows diminishing returns [1]
▶ More applications benefit from *hardware acceleration*
  - DSP, crypto, codecs, graphics, comm. protocols, etc.
  - Many more could benefit if hardware design wasn’t so *hard*
Hardware is growing

More specifically, hardware *acceleration*

- Miniaturization still has a decade to go [3]
  - Microarch. optimization shows diminishing returns [1]
- More applications benefit from *hardware acceleration*
  - DSP, crypto, codecs, graphics, comm. protocols, etc.
  - Many more could benefit if hardware design wasn’t so hard

- Hardware benefits more from *rigour*
  - Mass production, less *updateable*, lots of optimization
  - More error-prone, errors are more serious
Hardware design “status quo”

Myriad of languages for specific design tasks...

- **Simulation**: SystemC, VHDL/Verilog
- **Synthesis**: VHDL/Verilog, C/C++ (subsets)
- **Verification**: SAT solvers / Theorem provers
Hardware design “status quo”

Myriad of languages for specific design tasks...

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

The same situation in software seems bizarre nowadays:

- To “simulate” (interpret) your program, you use Haskell
- For compilation to x86, use C (non-standardized)
Functional hardware DSLs

- Solve **most** of the problem with multiple descriptions
- “Popular” example: Lava (Chalmers) (Ex.)
  - Description, simulation, testing in Haskell
  - Verification through external SAT solver
  - Non-modular proofs
- Also, Haskell types are not as strong as we want
  - \( \text{addN} :: \text{Int} \rightarrow ([\text{Bit}], [\text{Bit}]) \rightarrow [\text{Bit}] \)
Why Agda?
Dependent types

Why program using dependent types?

- Less runtime errors
  - More *correctness by construction*

- Reasoning about your programs
  - In the *same language* of the program itself
Dependent types for hardware

Better specification of sizing constraints

- **Haskell:** `addN :: Int -> ([Bit], [Bit]) -> [Bit]`
- **Agda:** `addN : (n : ℕ) → C (2 * n) (suc n)`

Rule out design mistakes

- Ex: short-circuits are ill-typed

Correctness proofs in the same language as the model
Syntax
Circuit syntax

- Low-level, *architectural* representation
- Untyped, but *well-sized*
Circuit syntax

- Low-level, *architectural* representation
- Untyped, but *well-sized*

```haskell
data C' : ℕ → ℕ → Set where
    Nil       : C' zero zero
    Gate      : (g# : Gates#) → C' (|in| g#) (|out| g#)
    DelayLoop : (c : C' (i + l) (o + l)) → C' i o

    Plug      : (f : Fin o → Fin i) → C' i o
    _≫_       : C' i m → C' m o → C' i o
    _≪_       : C' i₁ o₁ → C' i₂ o₂ → C' (i₁ + i₂) (o₁ + o₂)
    _|₁|₂_     : C' i₁ o → C' i₂ o → C' (suc (i₁ ⊔ i₂)) o
```

- No *floating* wires, no short circuits
Atomic types

► The whole Circuit module is parameterized by a record
  • Defining what is carried over the “wires”

► This Atomic class is similar to Haskell’s Enum
  • An atomic type needs to be finite
  • There’s a mapping between the type and \{0..n\}
  • “toEnum” / “fromEnum”

► Dependent types move runtime errors to compile-time:
  • Haskell: \texttt{succ maxBound} → runtime error
  • Agda: “\texttt{succ maxBound}” → doesn’t typecheck!
Atomic types (\textbf{Bool})

- Several interesting instances possible
  - \textbf{Bool}
  - \textbf{Int8, Int16, IntN}...
  - States of a state machine (any enumerated type)

- Simplest “useful”: \textbf{Bool}
  - We use the mapping $0 \to \text{False}; 1 \to \text{True}$
  - Order and choice of indices \textit{don’t matter}
Fundamental gates

- Circuits are built by combining smaller circuits
  - Ultimately, from a library of fundamental Gates

- 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{GateIdx} \rightarrow \mathbb{N} \\
\text{spec} : (g : \text{GateIdx}) \rightarrow (W (|\text{in}| g) \rightarrow W (|\text{out}| g))

- Dependent types help us again
  - The \text{GateIdx} type ranges in \{0..n\}
  - The function returned by \text{spec} works over words of the right size
Fundamental gates

- A “traditional” instance of Gates is BoolTrio
  - Set of gates: \{⊥, T, ¬, ∧, ∨\}
  - With the usual specification functions (from the stdlib)

- Other “interesting” instances:
  - Modular arithmetic
  - Cryptographic primitives
Putting all pieces together

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

\[ \lor C : C' \ 2 \ 1 \]
\[ \lor C = pFork \]
\[ \lor C \ (\neg C \ \mid \ pid \ \lor C) \ | \ (pid \ \mid \neg C \ \lor C) \]
\[ \lor C \]
Data abstraction

- Sometimes it’s more convenient to have typed circuit I/O
  - $\mathbb{C} (\text{Bool} \times \text{Bool}) \text{Bool}$ 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)

  $\downarrow : \alpha \rightarrow W i$

  $\uparrow : W i \rightarrow \alpha$
Semantics
Circuit semantics

- Our goal is to have two semantics:
  - Behavioural (done)
  - Structural (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 (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
  - \[
  \begin{array}{c}
  \text{⟦}_-\text{⟧} \to (c : \mathbb{C} i o) \{ p : \text{comb'} c \} \to (W i \to W o) \\
  \text{Example: } [\text{and} \text{]} (\text{true} :: \text{false} :: \varepsilon) \\
  \end{array}
  \]
Sequential simulation

- More general, for circuit with (possibly) internal state
  - Simulation takes a *stream* of inputs
- Modeled using Agda’s **Stream**
  - Type of infinite lists, defined by using *coinduction*
- User interface:
  - \([\_\_\_]^* : C' i o \rightarrow (\text{Stream } (W i) \rightarrow \text{Stream } (W o))\)
  - Example: \([\_\_\_] \text{ mapS not } [^* (\text{repeat } [\text{false }])\]
- Implementation detail
  - General **Stream** functions can “look into the future”
  - Our implementation uses only *causal stream functions*
Proofs
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”
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”

- We are particularly interested in *functional correctness*
  - Agreement with a *specification* on all inputs
  - Specification is a *function*
  - Example: \( \forall (x \ y : \text{Int8}) \rightarrow \llbracket \text{add}_{256} \rrbracket (x \ , \ y) \equiv x +_{256} y \)
Properties of circuit combinators

- Circuit combinators have *algebraic* properties
  - \_\_ (seq) is associative and has identity \texttt{pid} (monoid)
  - \_\_ (par) is also a monoid, with identity \texttt{Nil}
  - \( f \circ g \equiv \text{id} \rightarrow \text{Plug } g \} \text{Plug } f \equiv \text{pid} \)

- Agda is perfect for proving such statements
  - With a special notion of equality between circuits (\(\cong\))
  - *Equality up to simulation*
  - Equal behaviour \(\rightarrow\) opportunity for *optimization*

- \(\Pi\)-Ware can be used to define whole *classes* of circuits
  - With their own associated laws...
Current work

▶ 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 \texttt{scanl}

▶ General class + examples implemented in Π-Ware
  ▪ As M.Sc experimentation project (Yorick Sijsling)
  ▪ Proving associated laws in Agda
  ▪ Inspired by Ralf Hinze’s “An algebra of scans” [2]
Current work

- Correctness of sequential circuits
  - Temporal logic
Current work

► Correctness of sequential circuits
  • Temporal logic

► 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
Future
Future

“Proofs for free” for any specific circuit

- Given a collection of proofs, one for each possible input
- Generate the generalized statement
- Using reflection to make it all easy to use
"Proofs for free" for any specific circuit

- Given a collection of proofs, one for each possible input
- Generate the generalized statement
- Using reflection to make it all easy to use

Optimizations in generated VHDL

- Try to use circuit laws to justify "rewrite" steps
- Example: 

\[(a_1 \land a_2) \land a_3 \land a_4 \cong (a_1 \land a_2) \land (a_3 \land a_4)\]
Thank you!

Questions?

https://github.com/joaopizani/piware
References

Dark silicon and the end of multicore scaling.

Ralf Hinze.
An algebra of scans.
References II