Π-Ware: A Hardware Description Language embedded in Agda

Wouter Swierstra

Dept. of Information and Computing Sciences
Utrecht University

joint work with Joao Pizani Flor

Hardware design

- Low-level programming of complex tasks, often used in safety critical domains.
- In contrast to software, it's hard to release bugfixes – circuit designs *must* be correct

Hardware languages

VHDL & Verilog are the market leaders

- Some issues:
  - Language fragmentation: several languages and tools cobbled together.
  - Biggest divide: synthesis/sim. vs. formal verification

How can we use functional programming and dependent types in this domain?

Historical perspective

There are numerous domain-specific languages using functional programming technology to design and verify circuits.

Mary Sheeran's invited talk at ICFP last year gives a great overview.

One successful example: Lava (Bjesse et al.)

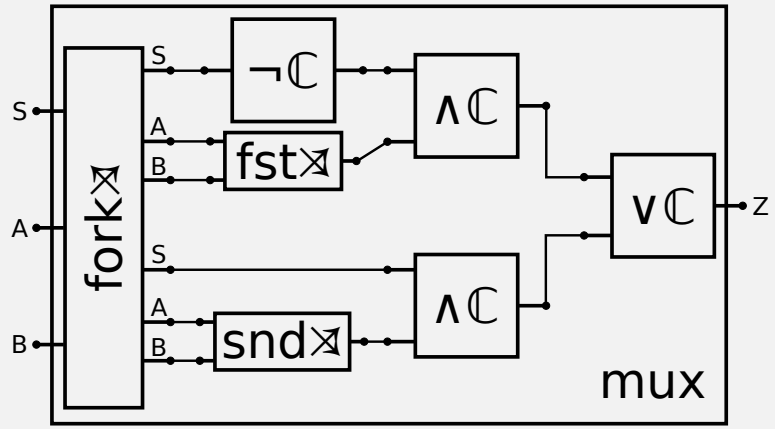
- Deep embedding in Haskell that relies on observable sharing.
- Clever combinators to define *circuit generators* and complex *connection patterns*
- Simulation and testing using tools like QuickCheck.
- Verification is done by calling an (automated) theorem prover.

Π-Ware

- An Agda EDSL for circuits
- **Very** low level of abstraction
  - A form of architectural combinator calculus
  - Trivial mapping to circuit schematics/netlists
- Advantages from Agda
  - Types ensure basic sanity conditions for circuits
  - A single language for the definition and verification of circuits.
- Our goal is not to replace the current generation of hardware languages, but rather to understand how our technology can be used to tackle this problem.

What a Π-Ware circuit looks like

- 2-to-1 multiplexer ("inverted" if-then-else circuit)
  - 3 inputs (S, A, B), 1 output (Z)
  - Boolean formula: $Z = (A \wedge \neg S) \vee (B \wedge S)$
- Schematic diagram



- Π-Ware model
  -

```
mux : ∀ {s} → ℂ {s} 3 1
mux = fork⨯⨯
   ⟫   (¬ℂ ‖ fst⨯₁ ⟫ ∧ℂ) ‖ (id⨯₁ ‖ snd⨯₁ ⟫ ∧ℂ)
   ⟫   ∨ℂ
```

Circuit datatype

- Deep-embedded DSL: a datatype for circuits ($\mathbb{C}$)

  $\mathsf{data}\ \mathbb{C} : \mathbb{N} \to \mathbb{N} \to \mathsf{Set}$

- Indexed by two natural numbers, corresponding to the number of input and output wires.
- We'll cover the constructors one by one.

Circuit constructors

Basic *gate*

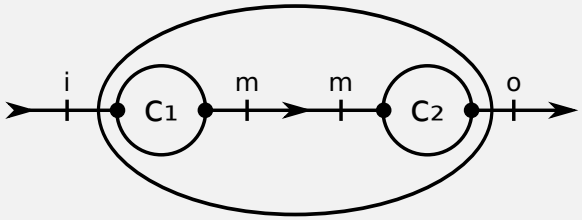
$$\frac{g : G}{\text{Gate } g : \mathbb{C}\ (\text{ins } g)\ (\text{outs } g)}$$

- Gate : $(g : G) \to \mathbb{C}\ (|\text{in}|\ g)\ (|\text{out}|\ g)$
- The language is parametrized by a gate type, G, corresponding to the fundamental building blocks of our circuits.
  - Each gate has a basic interface (|in|, |out|), counting the number of inputs and outputs;
  - Together with an associated behaviour
- Such gates may correspond to the familiar boolean operations, but you are free to choose your own.

Circuit constructors

Sequential composition

$$\frac{c_1 : \mathbb{C}\ i\ m \qquad c_2 : \mathbb{C}\ m\ o}{c_1 \gg\!\!\gg c_2 : \mathbb{C}\ i\ o}$$

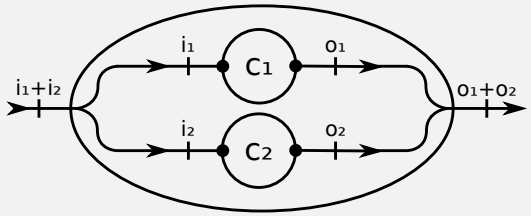


- Connects outputs of c_1 to inputs of c_2
  - $\_\gg\!\!\gg\_ : \mathbb{C}\ i\ m \to \mathbb{C}\ m\ o \to \mathbb{C}\ i\ o$
- The types ensure the interfaces line up.

Circuit constructors

Parallel composition

$$\frac{c_1 : \mathbb{C}\ i_1\ o_1 \qquad c_2 : \mathbb{C}\ i_2\ o_2}{c_1 \mid c_2 : \mathbb{C}\ (i_1{+}i_2)\ (o_1{+}o_2)}$$



- Passes different parts of the input to different subcircuits
  - $\_\|\_ : \mathbb{C}\ i_1\ o_1 \to \mathbb{C}\ i_2\ o_2 \to \mathbb{C}\ (i_1 + i_2)\ (o_1 + o_2)$

Circuit constructors

$$\frac{\mathsf{i}\ \mathsf{o} : \mathbb{N} \qquad \mathsf{f} : \mathsf{Fin}\ \mathsf{o} \to \mathsf{Fin}\ \mathsf{i}}{\mathsf{Plug}\ \mathsf{f} : \mathbb{C}\ \mathsf{i}\ \mathsf{o}}$$

- Explicit rerouting of wires
  Plug : $(\mathsf{Fin}\ o \to \mathsf{Fin}\ i) \to \mathbb{C}\ i\ o$
- This is used for swapping over wires, duplicating inputs, etc.
- Type *forbids* short-circuits (one output associated with multiple inputs) and floating wires (one output with no input).

Semantics

- As we have a deep embedding, we can define many different interpreters for our circuits: (translation to VHDL, computation of area, maximal delay, etc.)
- We'll focus on *functional* semantics
  $[\![\_]\!] : \mathbb{C}\ i\ o \to (\mathsf{Vec}\ \mathsf{Bool}\ i \to \mathsf{Vec}\ \mathsf{Bool}\ o)$
- Note: you can choose other atomic types than booleans if you want.

Functional stateless semantics

$[\![\_]\!] : \mathbb{C}\ i\ o \to (\text{Vec Bool}\ i \to \text{Vec Bool}\ o)$

- $[\![$ Gate g $]\!]$: Use function associated with g
- $[\![\ c_1 \ggg c_2\ ]\!]$: $[\![\ c_2\ ]\!] \circ [\![\ c_1\ ]\!]$
- $[\![\ c_1 \parallel c_2\ ]\!]$: uncurry _+_ $\circ$ map$\times$ $[\![\ c_1\ ]\!]$ $[\![\ c_2\ ]\!]$ $\circ$ splitAt i_1
- $[\![$ Plug f $]\!]$: each position p in the output, maps to $(f\ p)$ in the input

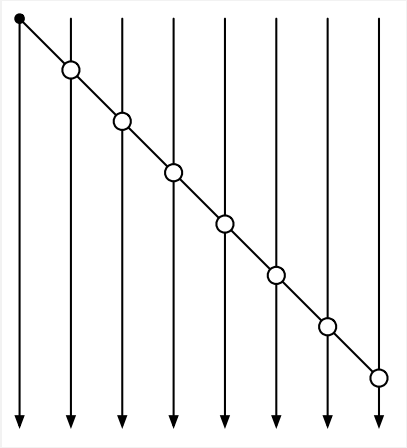
Parallel prefix circuits

We applied all of this in a case study on parallel prefix circuits, inspired by a paper by Ralf Hinze.

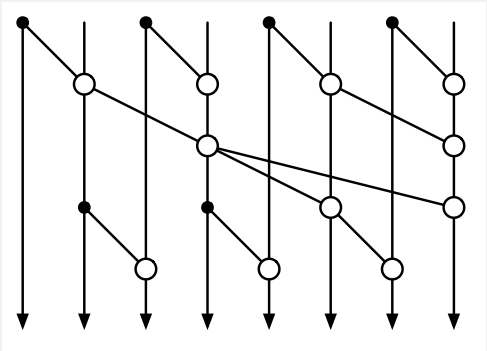
- *Parallel-Prefix Circuits* compute *scans*
  - Given: $(x_1, x_2, x_3, \ldots, x_n)$
  - Will compute $(x_1, x_1 \oplus x_2, x_1 \oplus x_2 \oplus x_3, \ldots, x_1 \oplus \cdots \oplus x_n)$
- When the binary operation $\_\oplus\_$ is associative, we have some flexibility on how to compute the subexpressions of $x_1 \oplus x_2 \oplus \cdots \oplus x_n$

Parallel-Prefix Circuits

- Example of PPC: serial scan



- Another example: minimal-depth scan



Can we show these two circuits are equivalent?

When are two circuits equivalent?

- When they display the same (functional) behaviour: given equal inputs, the circuits will produce equal outputs.
- However, if propositional equality as it is usually defined in Agda can only be used to compare two terms of the same type.
- We may define two circuit (generators) differently:

  $(c_1 : \mathbb{C}\ n\ (2 * n))$ and

  $(c_2 : \mathbb{C}\ n\ (n + n))$

  Despite the different types, they might have the same behaviour...

Vector equivalence

We can define a heterogeneous equivalence relation on vectors:

- two empty vectors are equivalent;
- two non-empty vectors are equivalent if their heads are propositionally equal and their tails are equivalent.

This corresponds to the usual propositional equality, but lets us at least discuss the possibility of two vectors of different types aspiring to be equal.

We consider two *circuits* equivalent, $(\_ \approx \_)$, when two equivalent input vectors, are mapped to equivalent output vectors.[1]

[1] There's a caveat...

Circuit properties

- Using this definition of $\_\approx\_$, we can prove some that our circuits have some basic algebraic structure:
  - Sequential monoid ($\_\gg\_$, $\mathsf{id}\!\!\nearrow$)
  - Parallel monoid ($\_\|\_$, $\mathsf{nil}\!\!\nearrow$)
  - Exchange law: $(c_1 \parallel c_2) \gg (d_1 \parallel d_2) \approx (c_1 \gg d_1) \parallel (c_2 \gg d_2)$

Parallel-Prefix Circuits

- We formalized/proven in Π-Ware:
  - What does it mean to compute a scan
  - All PPCs compute scans
  - Thus all PPCs are behaviourally equivalent
  - PPCs can be combined to form bigger PPCs
- Paper detailing Π-Ware and this case study
  - Submitted to the TYPES2015 workshop
  - `http://gitlab.com/joaopizani/piware-paper-2015`

Sequential and combinational circuits

The circuits we have seen so far are *combinational* – the result of the current clock cycle only depends on the current inputs.

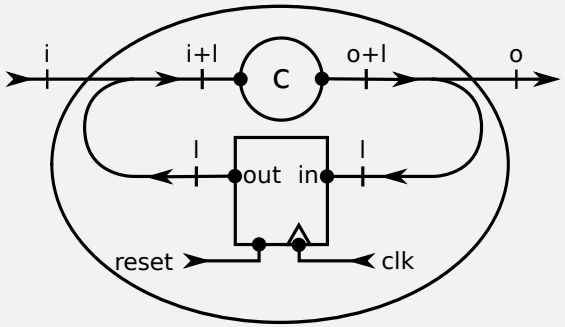
As soon as we allow loops in our circuits, things become more interesting.

Sequential circuits may store state.

A new circuit constructor

Loops

$$\frac{c : \mathbb{C}\ (i+l)\ (o+l)}{\text{DelayLoop} : \mathbb{C}\ i\ o}$$



- DelayLoop : $\mathbb{C}\ (i + l)\ (o + l) \rightarrow \mathbb{C}\ i\ o$
- This allows the next output to depend on *previous* inputs.

Semantics of sequential circuits

The type of our semantic function changes:

$$[\![\_]\!] : \mathbb{C}\ i\ o \to \mathsf{Stream}(\mathsf{Vec}\ \mathsf{Bool}\ i) \to \mathsf{Stream}(\mathsf{Vec}\ \mathsf{Bool}\ o)$$

The semantics of all our previous constructors can be lifted pointwise.

The loop case is more interesting...

Semantics of delay loops

In Haskell we might write something like:

```
run :: Circuit -> Stream (Word i) -> Stream (Word o)
run (Loop c) is =
    let (os,ss) = run c (is,0:ss)
    in os
```

This definition, however, is not obviously correct (structurally recursive/guarded corecursive) – and not accepted by Agda.

Instead, we require that the body of the loop is combinational.

Semantics of delay loops

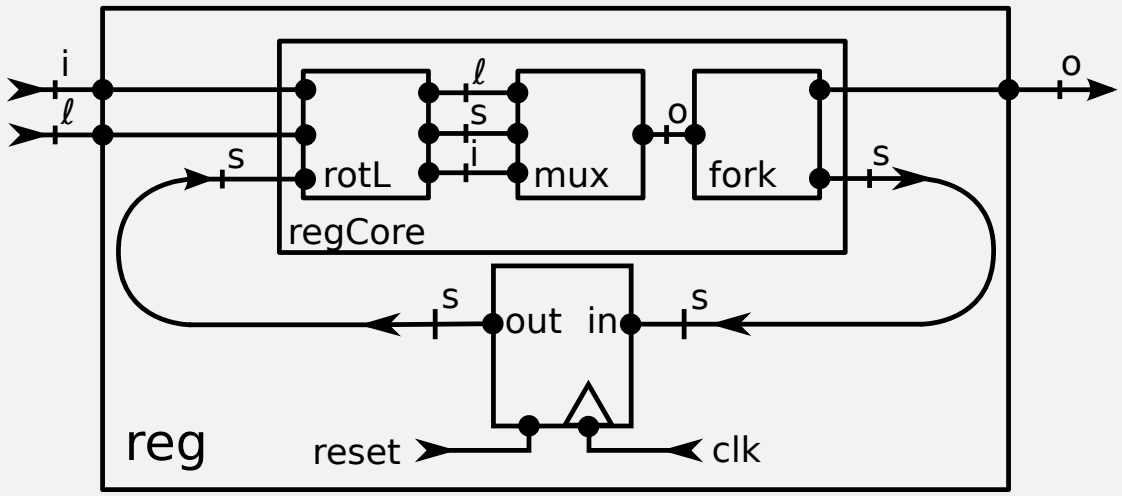
We revise our definition of the delay loop constructor:

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

The proof $\mathsf{isComb}\ c$ guarantees that we can simulate the loop body as a function between words (rather than streams of words).

This restriction allows us to give a suitable stream coalgebra, used to generate the stream of outputs.

Stateful circuit: reg



- A 1-bit register (1-bit input/output)
  - The state is also 1-bit wide
  - Core of the transition function: mux to *select*
- Π-Ware definitions:
  - $\mathsf{regCore} = (\mathsf{Plug}\ \mathsf{rotL}_3) \gg\!\rangle\ \mathsf{mux} \gg\!\rangle\ \mathsf{fork}_2$
  - $\mathsf{reg} = \mathsf{DelayLoop}\ \mathsf{regCore}$

By induction: regN

We can of course generalize reg to work on n bits.

- Π-Ware definitions:
  - regn−regs n = pars n reg
  - regn n = regn−distribute n ⟫ (eq_1 n ⟫ regn−regs n ⟫ eq_2 n)
- Annoying plugs to complete the definition:
  - regn−distribute: replicates and intersperses the "load" bit
  - **Equality** plugs: eq_1 and eq_2
    - Do not rearrange wires: no computational effect
    - "Convince" Agda that we can feed $(n + n)$ wires into a circuit that accepts $(n * 2)$ inputs

Coalgebraic semantics properties

- The coalgebraic semantics *extends* the stateless semantics
  - If $[\![\ c\ ]\!] = f$ then, $(\forall\ xs \to [\![\ c\ ]\!]\omega\ xs \approx \mathsf{map}\ f\ xs)$
- Composition properties:
  - Sequential
    - $[\![\ c \gg d\ ]\!]\omega\ xs \approx [\![\ d\ ]\!]\omega\ ([\![\ c\ ]\!]\omega\ xs)$
  - Parallel
    - $[\![\ c \parallel d\ ]\!]\omega\ (\mathsf{zip}\ \_{+}{+}\_\ xs\ ys) \approx \mathsf{zip}\ \_{+}{+}\_\ ([\![\ c\ ]\!]\omega\ xs)\ ([\![\ d\ ]\!]\omega\ ys)$

Some register properties

- We formalized the properties of registers, characterizing state the behaviour of registers (cf. Plotkin and Power *Notions of Computation Determine Monads*)
- Proven for both reg and regN

Future work

- We'd like to look into larger stateful case studies (CPUs, pipelining transformations, etc.)
- Higher-level typed language: shuffling about natural numbers is no fun.
- The plugs give a 'nameless' representation of variable – we'd like something a bit easier to use.
- Iteratively refine complex circuit specifications to more primitive gates.