Coquet: A Coq library for verifying hardware

João Paulo Pizani Flor

Department of Information and Computing Sciences, Utrecht University

June 13th, 2013

Universiteit Utrecht

Table of Contents

Intro
What is it
Why is it useful
Technical overview
Diving into Coquet
Interfaces
Circuit type
Semantics
Applied example
A half-adder
Conclusions

Section 1

Intro

What is Coquet

A Coq *library* for hardware specification and verification.

What is Coquet

A Coq *library* for hardware specification and verification.

Library
: Datatypes, typeclasses, instances, combinators, proofs, etc.

Deep-embedded
: *Circuit* datatype, structurally defined.

Dependently-typed
: The *circuit* AST is fully-typed.

Provably correct
: Circuits can be proven to *realize* a specification.

Why is Coquet useful

- Coq is more expressive than other theorem provers (HOL, ACL2, etc.).
- Dependent types help catch common errors soon in the design process.
- Deep-embedding brings interesting possibilities:
  - Functions that transform circuits (can be proven correct).
  - Simulating *and synthesizing* circuits (netlists).

Section 2

Technical overview

Circuit interface

- The *interface* of a circuit is the set of its input and output ports.
- Modeled in Coquet as *parameters* of the circuit datatype.

```
example : Circuit n m
```


Universiteit Utrecht

Circuit interface

- The *interface* of a circuit is the set of its input and output ports.
- Modeled in Coquet as *parameters* of the circuit datatype.

```
example : Circuit n m
```

- The parameters n and m can be of any *finite* type
  - More details on these finite types later...
  - For example, disjoint sums of "units":

```
nBitAdd : Circuit (U :+: sumn U n :+: sumn U n)
                  (sumn U n :+: U)
```


Universiteit Utrecht

Defining circuits with combinators

Circuits are modeled *hierarchically* with combinators, closely mimicking the “pen-and-paper” approach.

Defining circuits with combinators

Circuits are modeled *hierarchically* with combinators, closely mimicking the "pen-and-paper" approach.

Circuits are built using:

- Any of a set of *fundamental* components, user-defined.
- Serial composition.
- Parallel composition.
- Re-arranging and re-ordering of ports, done by *plugs*.
- Loop combinator, to create feedback loops.

Again, more details soon...

Re-arranging wires with plugs

As the combinators connect the circuits in a "nameless" fashion, we need rewiring circuits, called *plugs*

- The type of a plug that duplicates its input COULD be:
  `dup : Circuit U (U :+: U)`
- To *construct* a plug, we need a function mapping *outputs* to *inputs*. More details later.

Circuit semantics

Until now, we have only dealt with the *syntax* (the structure) of circuits.

- The meaning or *semantics* of a circuit (`x : C n m`) is defined as a relation between its inputs and outputs, where:
  - *inputs* is a function of type $(n \to \mathbb{T})$.
  - *outputs* has type $(m \to \mathbb{T})$.
  - $\mathbb{T}$ is the type of what is carried in the wires.
- The relation is defined *inductively* on x, and denoted as follows: $x \vdash^n_m \text{ins} \bowtie \text{outs}$.

Section 3

Diving into Coquet

Tagged unit types

Remember that Coquet uses arbitrary *finite* types for the inputs and outputs of a circuit.

- One way to build a finite type is by summing some *units*.
- This is inconvenient and confusing at *least*.
  - To allow easily discernible input/outputs, Coquet uses *tags*:

```
Inductive tag (t : string) : Type := _tag : tag t
```

  - For example:

```
halfAdder : Circuit (_tag "in1" :+: _tag "in2")
                    (_tag "sum" :+: _tag "cout")
```

- Also, this is not the only way to have finite types. . .

The Finite typeclass

In general, any type with an instance for the `Fin` typeclass can be used as input or output in circuits.

- A typeclass in Coq has approx. the same meaning as in Haskell.
  - Predicate/relation over/among type(s).
  - Instances are named and passed.
  - May contain (of course) proofs of properties over types.
- The `Fin` typeclass looks like this:

```
Class Fin A := {
    eq_fin : eqT A;
    enum   : list A;
    axiom  : forall (x : A), count (equal x) enum = 1
}.
```

- Instances for unit, tagged unit, sum, etc.

The Circuit datatype

- The (dependent) type of circuits looks like:

```
Context {tech : Techno}
Inductive Circuit : Type -> Type -> Type :=
| Atom : forall {n m : Type} {Fn : Fin n} {Fm : Fin m},
    techno n m -> Circuit n m
| Plug : forall {n m : Type} {Fn : Fin n} {Fm : Fin m},
    (f : m -> n) -> Circuit n m
| Ser : forall {n m p : Type},
    Circuit n m -> Circuit m p -> Circuit n p
| Par : forall {n m p q : Type},
    Circuit n p -> Circuit m q -> Circuit (n :+: m) (p :+: q)
| Loop : forall {n m p : Type},
    Circuit (n :+: p) (m :+: p) -> Circuit n m
```

- Parameterized by the type of fundamental gates:

```
Class Techno := techno : Type -> Type -> Type.
```


Universiteit Utrecht

Plugs, plugs, plugs. . .

As seen, a Plug requires a function from `m` to `n`.

```
Inductive Circuit : Type -> Type -> Type :=
| Plug : forall {n m : Type} {Fn : Fin n} {Fm : Fin m},
    (f : m -> n) -> Circuit n m
```

- Let's examine again the circuit which duplicates its input:

```
Plug (fun x => match x with
               | inl e => e
               | inr e => e
               end).
```

- In simple enough cases (like this one), proof-search suffices.

Meaning relation over circuits

As already said: The semantics of a circuit is given by a relation between its inputs and outputs, defined inductively.

- The base case involves defining instances of the Techno class for a given set of basic gates:

```
Class Technology_spec (tech : Techno) T :=
    spec : forall {a b : Type},
    tech a b -> (a -> T) -> (b -> T) -> Prop
```

- The meaning relation is generated using this parameter and other rules (one for each constructor of Circuit).

Realizing a spec, implementing a function

- The meaning relation is not a specification for a circuit.
  - It is a strict and detailed definition of behaviour.
  - *Too* detailed, in fact...
- Coquet relies on 2 typeclasses to let the user *specify* a circuit:

```
Context {n m N M : Type}
        (Rn : Iso (n -> T) N) (Rm : Iso (m -> T) M)
Class Realise (c : Circuit n m) (R : N -> M -> Prop) :=
    realise : forall ins outs,
        Meaning c ins outs -> R (iso ins) (iso outs)
Class Implement (c : Circuit n m) (f : N -> M) :=
    implement : forall ins outs,
        Meaning c ins outs -> iso outs = f (iso ins)
```

- Proving the spec is done by providing instances of these classes (proof objects).

Section 4

Applied example

Case studies defined in the paper

- Half-adder
- Full 1-bit adder
- Ripple-carry n-bit adder
- Divide-and-conquer n-bit adder
- Sequential circuits (memory elements)

Here we'll only take a peek at a half-adder...

Definition of a half-adder circuit

(Nice diagram on the whiteboard. . .)

```
Context a b s c : string (* section variables *)

Definition HADD :
    Circuit (_tag a :+: _tag b) (_tag s :+: _tag c) :=
    Fork2 (_tag a :+: _tag b) |> (XOR a b s & AND a b c).
```

Proving a half-adder circuit

- We must prove that HADD implements the function:

```
Definition hadd : (bool, bool) -> (bool, bool) := fun x =>
    match x with
    | (a, b) => (a xor b, a /\ b)
    end.
```

- Our claim that HADD implements hadd is written as:

```
Instance HADD_Spec : Implement
    (* iso on inputs *)
    (* iso on outputs *)
    HADD hadd.
Proof.
    ...
```

- Then we need to prove it (goal on whiteboard. . .).

Section 5

Conclusions

Comparison with related works

- Circuits were already modeled in the HOL theorem prover, but using a shallow embedding.

Comparison with related works

- Circuits were already modeled in the HOL theorem prover, but using a shallow embedding.
- In Coq, also only a shallow embedding had already been attempted.

Comparison with related works

- Circuits were already modeled in the HOL theorem prover, but using a shallow embedding.
- In Coq, also only a shallow embedding had already been attempted.
- In Haskell, there was Lava, in which verification is reduced to finite-sized instances of circuits.
  - In contrast, in Coq we prove the correctness of parametric circuits, for any size of inputs.

Final words

- "Programming" hardware in functional languages has lots of advantages, and using dependently-typed languages has even more.

Universiteit Utrecht

Final words

- "Programming" hardware in functional languages has lots of advantages, and using dependently-typed languages has even more.
- Proving the correctness of hardware is an awesome idea.
  - Even better if you can synthesize it to an FPGA or even ASIC.

Final words

- "Programming" hardware in functional languages has lots of advantages, and using dependently-typed languages has even more.
- Proving the correctness of hardware is an awesome idea.
  - Even better if you can synthesize it to an FPGA or even ASIC.
- There is still some polishing to be done:
  - Couldn't make the lib compile in Coq 8.4.
  - There seems to be a synthesis facility, but I'm not sure...
  - Have a "standard library" of basic circuits.

Questions?