Comparing functional Embedded Domain-Specific Languages for hardware description

João Paulo Pizani Flor
Department of Information and Computing Sciences,
Utrecht University - The Netherlands
e-mail: j.p.pizaniflor@students.uu.nl

Wednesday 20th November, 2013

1 Introduction

Hardware design has become nowadays an activity which is, to say the least, extremely complex, if not error-prone. The sheer size of circuits has increased. Also, low-level concerns such as power consumption, error correction, parallelization and layout in general have to be incorporated earlier and earlier in the design process, breaking modularity and making it harder to validate and verify the correctness of circuits.

In this context, researchers have been suggesting for a long time (since the 1980s, in fact) the usage of functional programming languages to model circuits. One particular line of research is to create Embedded Domain-Specific Languages for hardware description based on existing functional programming languages, such as Haskell.

There is a multitude of EDSLs for hardware description out there, but they vary wildly and on a number of aspects: host language, level of abstraction, capabilities of simulation, formal verification, synthesis (generation of netlists) and integration with other tools, to name a few. All this variety can make the task of choosing a hardware EDSL for the task at hand extremely daunting and time-consuming.

The main goal of this experimentation project is to establish some order in this landscape, and to perform a practical analysis of some representative functional hardware EDSLs. By reading the materials produced in this project (circuit models, test cases, generated netlists, report), a hardware designer wishing to use a functional hardware EDSL for his next design should gain some insight about the strengths and weaknesses of each language and have an easier time choosing one.

As an additional result of this research, we intend to identify recent, cutting-edge developments in the Haskell language and its implementations that the analyzed EDSLs could benefit from. Also, we intend to discuss to which extent some shortcomings of the EDSLs could be overcome by having them hosted in a dependently-typed language.

2 Methodology

In this project, we compared a number of functional hardware EDSLs that we considered representative (more details on the choice of EDSLs further ahead). The comparison was performed on a number of *aspects* for each EDSL, and the analysis was done by considering a *sample set of circuits* used as case studies.

We tried to model all circuits in all EDSLs considered, and as similarly as possible in all of them. To avoid using any of the analyzed EDSLs as "base", we provide a neutral, behavioural description of the circuits.

2.1 The languages

The embedded Hardware Description Languages we decided to analyze are:

Lava The Lava[1] language, developed initially at Chalmers University in Sweden. Lava is deeply embedded in Haskell, and provides features such as netlist generation and circuit verification using SAT-solvers. There are several "dialects" of Lava available, and the one used for this project is considered the "canonical" one, originally developed at Chalmers.

ForSyDe The Haskell ForSyDe library is an EDSL based on the "Formal System Design" approach[4], developed at the swedish Royal Institute of Technology (KTH). It offers both shallow and deep embeddings, and provides a significantly different approach to circuit modelling, using *Template Haskell* to allow the designer to describe combinational functions with Haskell's own constructs.

Coquet The Coquet[2] EDSL differs from the other 2 mainly because it's embedded in a dependently-typed programming language (the Coq theorem prover). Coquet aims to allow the hardware designer to describe his circuits and then *interactively* prove theorems about the behaviour of whole *families* of circuits (using proofs by induction).

The Lava EDSL has several "dialects", among which are Xilinx-Lava, York-Lava, Kansas-Lava and Chalmers-Lava. Xilinx-Lava was developed by Satnam Singh and puts a greater emphasis on the *layout* of the described circuits, focusing on their implementation in Xilinx's FPGAs. York-Lava was developed as part of the Reduceron project, and is a variation of Chalmers-Lava, omitting some features and adding some others, like a "Prelude" of commonly used circuits ((de)multiplexers, (de)coders, etc.), RAM memory blocks, among others. Chalmers-Lava is considered the "standard" dialect, also being the one which was first developed.

2.2 The aspects evaluated

For each of the hardware description EDSLs we experimented with, a number of *aspects* were evaluated. The evaluated aspects do not necessarily make sense for *all* EDSLs, therefore our presentation follows a language-centric approach, in which we expose the strenghts and weaknesses of each EDSL concerning the applicable aspects.

Without further ado, the following aspects are considered in the analysis:

Simulation The capability of simulating circuits modeled in the EDSL (and the ease with which it can be performed). Simulation is understood in this context as *functional* simulation, i.e, obtaining the outputs calculated by the circuit for certain input combinations.

Verification The capability of verifying *formal properties* concerning the behaviour of circuits (and the ease with which verification can be performed). The properties we are interested in are those which are *universally quantified* over the circuit's inputs. As an example of such a property, we might have:

$$\forall a \forall b \forall \text{sel}\, (\text{MUX}(a, b, \text{sel}) = a) \vee (\text{MUX}(a, b, \text{sel}) = b)$$

Genericity Whether (and how well) the EDSL allows the modelling of *generic* or *parameterized* circuits. An example of a generic circuit is a multiplexer with 2 n-bit inputs and 1 n-bit output, or a multiplexer with n 1-bit inputs and 1 1-bit output. Besides parametrization in the *size* if inputs and outputs, we will also analyze whether the EDSL provides chances for parametrization on other functional and/or non-functional attributes.

Depth of embedding Whether the EDSL models circuits with a *shallow* embedding (using predicates or functions of the host language), a *deep embedding* (in which circuits are members of a dedicated data type), or anything in between. The depth of embedding of an EDSL might have consequences for other aspects being analyzed.

Integration with other tools How well does the EDSL allow for interaction with (getting input from / generating output for) other tools in the hardware design process. For example, synthesis tools for FPGAs or ASICs, timing analysis tools, model checkers, etc.

Extensibility The extent to which the user can *add* new interpretations, data types, and combinator forms to the language. For example, the user might want to model circuits that consume and produce custom datatypes, or might be interested in extracting *metrics* from a circuit such as power consumption, number of elementary gates, etc.

3 Modeled circuits

When thinking of which circuits to model using the analyzed EDSLs, some principles guided us. First of all, they shouldn't be too simple but also not too complex. Some very simple circuits (adders, counters, etc.) are very often shown as examples in the papers that define the EDSLs themselves, as well as in tutorials. On the other hand, we also did not want to model too complex circuits; that would require too much effort on the hardware design itself, and diverge from the focus of this project, which is to evaluate and analyze the EDSLs.

Another principle that guided our choice is that the circuits should be immediately familiar to anyone with some minimal experience in hardware design. We avoided, therefore, considering application-specific circuits such as those for Digital Signal Processing (DSP), implementing communication protocols, etc. Having ruled out this class of circuits, we were left to choose from circuits that form a general-purpose computing machine, such as arithmetic units, memory blocks, control units and so forth.

Finally, we wanted to choose among circuits that already had a well-defined, *behavioural* description, to avoid using any of the analyzed EDSLs as "basis" of comparison.

Taking these considerations into account, we chose to implement, in each of the EDSLs analyzed, three circuits originating from the book "The Elements of Computing Systems"[3]. This book aims to give the reader a deep understanding of how computer systems work by taking a hands-on approach, in which the reader is given the most basic logic gates and builds, step-by-step, all the hardware and software components necessary to implement a complete computer system.

From the hardware design part of the book, we took our three circuits to be modeled:

- A simple Arithmetic Logic Unit (ALU), from here onwards referred to as "circuit 1".
- A RAM memory block with 64 words, from here onwards referred to as "circuit 2".
- A CPU with an extremely reduced instruction set (capable of executing the *Hack* assembly language defined in the book) from here onwards referred to as "circuit 3".

3.1 The ALU circuit

The Arithmetic Logic Unit built by us is a 2-input ALU, in which each of the inputs (as well as the output) is a 16-bit long word (interpreted as an integer in two's-complement encoding). It is capable of computing several functions, and the choice of which function to compute is made by setting the ALU's 6 *control bits*. To become more familiar with this circuit, let's first take a look at its block diagram, shown in figure 1

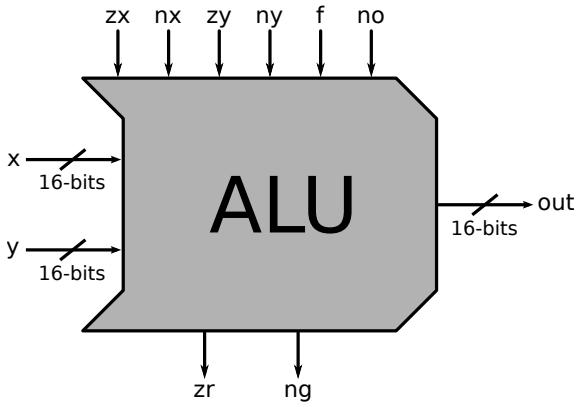

Figure 1: Block diagram of circuit 1, showing its input and output ports.

Each of the 6 control bits to the ALU has, in isolation, a well-defined effect on the inputs or outputs to the ALU core. The bits `(zx, nx, zy, ny)` control "pre-processing" steps for the inputs `x` and `y`, with the following behaviour:

zx and zy *Zeroes* the x input (respectively y). The ALU core will receive `0` as input.

nx and ny Performs *bitwise negation* of input x (respectively y).

Therefore, the ALU "core" itself (adder, and gate) has, as inputs, the results of performing these pre-processing steps controlled by `(zx, nx, zy, ny)`. Furthermore, the *output* of the ALU core can also be *bitwise negated* as a "post-processing" step, controlled by bit `no`.

Finally, the control bit f can be used to select which operation is to be performed by the ALU core: if we wish to add the two inputs, we need to set $f = 1$, and if we want bitwise conjunction, then we need to set $f = 0$.

Besides the main (16-bit wide) output of the ALU, there are two other output *flags*, that indicate predicates over the main output:

zr Is high whenever $out = 0$.

ng Is high whenever $out < 0$.

With the ALU in the context of a microprocessor, these flags can be used, for example, to facilitate conditional jumps.

Even though there are $2^6 = 64$ possible combinations for the values of the control bits, only 18 of these combinations result in interesting functions – that because several combinations of control bits can be used to calculate the same function. We show these 18 functions that the ALU can calculate on table 3.1.

zx	**nx**	**zy**	**ny**	**f**	**no**	**out=**
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	1	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	$\neg x$
1	1	0	0	0	1	$\neg y$
0	0	1	1	1	1	$-x$
1	1	0	0	1	1	$-y$
0	1	1	1	1	1	$x + 1$
1	1	0	1	1	1	$y + 1$
0	0	1	1	1	0	$x - 1$
1	1	0	0	1	0	$y - 1$
0	0	0	0	1	0	$x + y$
0	1	0	0	1	1	$x - y$
0	0	0	1	1	1	$y - x$
0	0	0	0	0	0	$x \wedge y$
0	1	0	1	0	1	$x \vee y$

Table 1: Functions that the ALU can calculate, given different settings of the control bits

3.2 The RAM circuit

Circuit 2 is a block of RAM with 64 lines and in which each line is a 16-bit word. Actually, using the term "RAM" to refer to this component is an abuse of terminology, as this circuit is nothing more than a register bank.

All the input and output ports of the circuit are pictured in its block diagram, shown in figure 2.

The circuit has one 16-bit output, named "out", and three inputs ("in", "address" and "load"). The "in" port is 16-bit wide and holds a value to be written into the RAM. The "address" port has a width of $\log_2 64 = 6$ bits and holds the address in which reading or writing is to be performed. Finally, the 1-bit "load" input controls whether the value currently at "in" should be written to the selected address. There is also one *implicit* input for a clock signal in this component. Implicit, in this case, means that the clock signal is not present in any of the models that we developed for this circuit, but must be present at any physical implementation.

The temporal behaviour of this memory block is as follows: At any point in time, the output "out" holds the value stored at the memory location specified by "address". If the "load" pin is high, then the value at "in" is loaded into the memory word specificied by "address". The loaded value will then be emitted on the output at the **next** clock cycle.

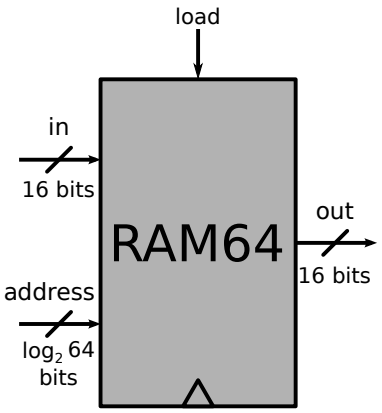

Figure 2: Block diagram of circuit 2, a RAM of 64 lines

3.3 The Hack CPU circuit

Circuit 3, the largest and most complex circuit among the ones we have chosen to implement, is the Central Processing Unit for the *Hack* computer, the machine described in the book "The Elements of Computing Systems"[3].

The Hack computer is based on the *Harvard architecture*, that means that it has different storage component and signal pathways for instructions and data. Therefore, the Hack CPU expects to be connected to *two* memory blocks, the instruction memory and the data memory. Having this in mind facilitates the understanding of the CPU's block diagram, shown in figure 3

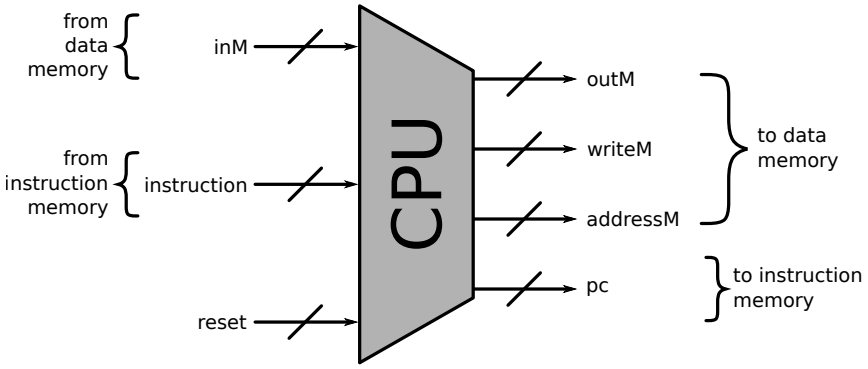

Figure 3: Block diagram of circuit 3, the Hack CPU

The Hack architecture has an *extremely* reduced instruction set, and consists in fact of only two instructions (each 16-bit wide): A (meaning "address") and C (meaning "compute"). The A instruction can be used as a means to load numerical literals into the data memory, as well as setting a special "cache" register inside the CPU. The C instruction is the one responsible for effectively performing computations using the ALU, testing outputs and jumping. More details about programming in the Hack assembly language can be found in [3].

The meaning of each of the CPU's input and output ports becomes much clearer when we look at the context in which the CPU is inserted, namely, the memory modules to which it is connected. So, let's analyze the CPU's ports by taking a look at figure 4.

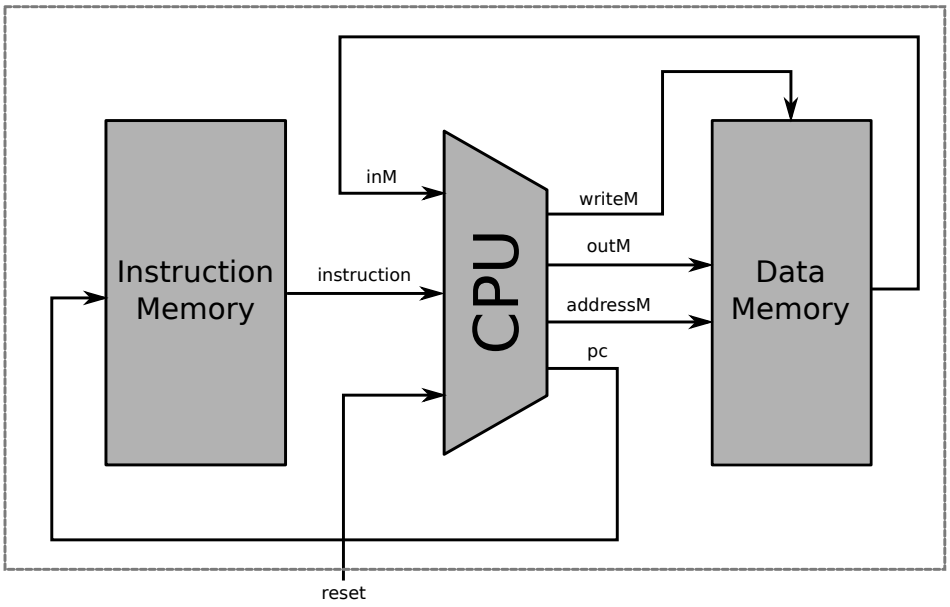

Figure 4: The Hack CPU connected to the data and instruction memory blocks

4 Analysis of the EDSLs

4.1 Lava

The Lava EDSL...

4.2 ForSyDe

The Haskell ForSyDe library is an implementation of the "Formal System Design" approach to hardware modelling[4]. The ForSyDe approach per se has several significant differences when compared to Lava, and even when the two EDSLs agree on *what to do*, sometimes they differ on how to achieve those goals.

To better understand what characterizes the ForSyDe methodology, we first have to establish some vocabulary:

System In ForSyDe, a system or circuit is a network of processes interconnected by *signals*.

Signal A signal is, intuitively, a stream of information that flows between processes. More concretely, it carries events of some type, and each event has an associated tag. The meaning of the tag is defined by the *model of computation*4.2.1 used.

Process A process is nothing more than a pure function on signals. A process *is able* to hold internal state. But, given the same input (possibly infinite) signals, it produces the same output signals.

Process constructor Every ciruit in ForSyDe, even simple combinational ones, is built with a *process constructor*. A process constructor can be seen as a skeleton of behaviour, and it clearly separates computation from synchronization aspects. A process constructor takes a combinational function (called *process function*) as parameter – expressing the computation aspect of the process, and possibly some extra

values. There are combinational and sequential process constructors, and some representative examples from each class will be described in more detail in a specific subsection 4.2.2.

4.2.1 Models of Computation

The definition of signal given above is purposefully "vague" mainly because the precise definition of signals is given by the *model of computation* being used. ForSyDe has (currently) process constructors for the following models of computation:

Synchronous Lorem ipsum

Untimed Lorem ipsum

Continuous Lorem ipsum

Among those, perhaps the most "notable" one is the Synchronous MoC, because it reflects the usual interpretation of signals as wires and the vast majority of digital designs nowadays having a global clock. Also, all of our studied circuits were modelled in ForSyDe using the Synchronous MoC. Therefore, it is interesting to take a deeper look at it.

First, lets take a look at the behaviour of a system which has an one integer input port and one integer output, and in which the value of the output is equal to the input plus 4. The interface and internal architecture of this system (*addFour*) is depicted in figure 5.

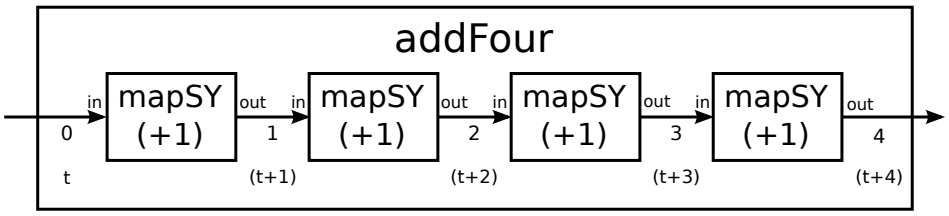

Figure 5: The addFour circuit, example of usage of the Synchronous MoC

In this system, each of the constituent processes is built using the `mapSY` process constructor, a constructor of the synchronous model of computation (its name ends in "SY"). It takes a combinational function (in this case "+1") and evaluates it for each event in the input signal, generating a corresponding event in the output signal.

TODO: Talk about shallow and deep embedded signals (deep only supports synchronous).

4.2.2 Synchronous Process Constructors

4.2.3 Circuits modeled

TODO: expose and discuss the circuits we modeled as case study, highlighting the advantages and disadvantages of ForSyDe found while modelling them.

Advantages: more modular VHDL generated (hierarchical). Higher-level descriptions.

Disadvantages: name handling, not actively maintained

4.3 Coquet

The third analyzed EDSL for hardware description, Coquet[2], is strikingly different from both others. Most of these differences can be explained in one way or another by its choice of host "language" – Coq.[1]

[1] "Coq" is not the name of a language, but a theorem-proving system that uses different languages for defining terms, interactive commands, and user-defined tactics

Coq is an interactive theorem prover based on *intuitionistic type theory*. In the context of Coq, the concepts of "term" and "type" are far more intertwined than, say, in Haskell. Types in Coq can contain references to terms and vice-versa. A very typical example of these so-called *dependent types* is the type(-family) of vectors with a certain length:

```
Inductive vec A : nat -> Type :=
    | nil  : vec A 0
    | cons : forall (h : A) (n : nat),  vec A n -> vec A (S n).
```

By having the length of the vector being part of the type, we can *enforce* several useful properties of functions operating on vectors. In fact, the type-system of Coq is so expressive that it can encode any proposition of *intuitionistic propositional logic*, a formal logic in which almost all of mathematics can be proven.

Given such expressive power, one can imagine that it might be useful to express circuits in Coq, and use it to prove interesting properties about these circuits. This is exactly the goal of Coquet. How this goal is achieved and the modelling of our studied circuits in Coquet is discussed in the following subsections.

4.3.1 Modelling circuits

Coquet is a deep-embedded DSL, thus it represents circuits as a datatype. By using dependent types, a designer modelling a circuit in Coquet is able to prevent certain kinds of "errors" much earlier in the design process, because the *well-formedness* is guaranteed by construction, i.e, every circuit built using the constructors provided by Coquet are well-formed by definition. Let's take a look at the *Circuit* data type definition:

```
Context {tech : Techno}
Inductive ℂ : Type -> Type -> Type :=
| Atom : ∀ {n m : Type} {Hfn : Fin n} {Hfm : Fin m}, techno n m -> ℂ n m
| Plug : ∀ {n m : Type} {Hfn : Fin n} {Hfm : Fin m} (f : m -> n), ℂ n m
| Ser : ∀ {n m p : Type}, ℂ n m -> ℂ m p -> ℂ n p
| Par : ∀ {n m p q : Type}, ℂ n p -> ℂ m q -> ℂ (n + m) (p + q)
| Loop : ∀ {n m p : Type}, ℂ (n + p) (n + p) -> ℂ n m
```

Figure 6: The Circuit (ℂ) datatype in Coquet

The ℂ type is parameterized by two types. These types are the *input* and *output* types of the circuit, respectively. They **do not** represent what is "carried" on the wires, but the *structure* of the circuit's input and output ports: How many of them there are, how they grouped and how they are named.

There are 2 *atomic* constructors from which an element of ℂ can be built and 3 *combinators*, which build a circuit based on other circuit(s). By observing the serial and parallel composition combinators (`Ser` and `Par`, respectively), we notice that the input and output types match exactly as expected.

The 2 atomic constructors constrain the types `n` and `m` by requiring them to have instances of the "Fin" type class, i.e, they have to be *finite* types (types from which a finite list of unique elements can be obtained). This constraint is important given the interpretation that these types (`n` and `m`) have: each element of `n` (respectively `m`) stands for an input (respectively output) "wire" in the circuit interface.

The case of the "Atom" constructor is particularly revealing of how Coquet works: this constructor is parameterized by an *instance* of the type class `Techno` for the types `n` and `m`. What this instance provides (in the code fragment that reads "`techno n m`") is the *type of*

the fundamental gate in the technology being used. We could choose our modeled circuits to have, for example, NAND, NOR, or other (more exotic) gates as fundamental.

As an "usage example" of Coquet, we show two simple circuits (`NOT` and `HALFADD`), along with proofs that they implement the expected functions over booleans. Let's start with `NOT`:

```
Definition NOT x nx : ℂ [:x] [:nx]  :=  Fork2 _ |> (NOR x x nx).

Instance NOT_Implement {x nx} : Implement (NOT x nx) _ _ negb.
Proof.
    intros ins outs H.
    unfold NOT in H.
    tac.
Qed.
```

The input type of `NOT` is a *tagged unit* with tag `x`, similarly, the output type has tag `nx`. There is some *notation* introduced by Coquet to make the creation of tagged units more convenient. The `NOT` circuit is a serial composition (denoted as `|>` of a `Fork2` circuit (which simply splits the input into two identical copies) and a `NOR` circuit, which is the underlying fundamental gate in this case.

Below the definition of the circuit itself we state and prove the fact that our circuit implements the desired function (boolean negation, `negb`). More details on exactly what is meant by "implements", as well as the workings of plugs and tags, are exposed further ahead. Now let's take a look at a half-adder described in Coquet:

```
Definition HADD a b s c : ℂ ([:a] + [:b]) ([:s] + [:c]) :=
    Fork2 ([:a] + [:b]) |> (XOR a b s & AND a b c).

Instance HADD_Implement {a b s c} :
    Implement (HADD a b s c) _ _
    (fun (x : bool * bool) => match x with (a,b) => (xorb a b, andb a b) end).
    Proof.
        unfold HADD;  intros ins outs H;  tac.
    Qed.
```

First of, the sum types that are given as parameters to ℂ indicate that we have two input ports and two output ports. By using `Fork2` on a binary sum (`[:a] + [:b]`), we create as output a sum type in which each of the components is in itself a sum: that matches exactly the interface of the component after the `|>` operator. On the right side of the serial composition, we have a parallel composition of `XOR` and `AND`, giving two outputs: respectively the sum (`[:s]`) and carry-out (`[:c]`)

Together with the definition, we prove that the `HADD` circuit implements the boolean function we would expect, and the proof is similar to the case of `NOT`. It makes use of a Coquet custom tactic (`tac`), but a more throughout example of proof of functional correctness will be given futher ahead. Also, in the case of the adder used in our case study (ripple-carry adder), we prove that the circuit implements the *actual* addition function on binary integers, and not some boolean equivalent.

As a last detail on the "user interface" of Coquet, there is the definition of what exactly are the input and output types (a circuit of type `ℂ n m` has input type `n` and output type `m`). Usually, in the Coquet paper[2] and in the examples provided with the library, input and output types are *sum types* in which the terms of the sum are *tagged units*. Using tags serves a form of "documentation", giving someone reading the circuit model an idea of what role does each input/output port play. The *type family* of tagged unit types is defined as follows:

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

Notation "[: x ]" := (tag x).
Notation "[! x ]" := (_tag x).
Notation "[!!]"   := (_tag _).
```

For each string `t`, there is a type `tag t`, and this type has exactly one inhabitant. There are also, as part of Coquet, some definitions to make working with sum types less tiresome. For example, there is the function `sumn` which, given a type `t` and a natural `n`, returns a sum type with `n` elements and in which each element has type `t`. This might be useful if we are defining a n-bit adder:

```
Definition RIPPLE cin a b cout s n :
    ℂ  ([:cin] + sumn [:a] n + sumn [:b] n)  (sumn [:s] n + [:cout]) := ...
```

While the provided examples use sums of tagged units as the input/output types, they can be more general: as seen in the definition of the circuit type (Fig. 6), the only requirement is that they belong to the `Fin` type class, which is defined as follows:

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

4.3.2 Circuit semantics in Coquet

4.3.3 Possible improvements and future work

The approach to circuit modelling is *very* generic, and leaves room for extension in several aspects, even though the library is already provided with some reasnoable deafults.

For example, the definition of the *meaning relation* for circuits (by induction on circuit structure) is parameterized by the *type* `T` *of was is carried in the wires*. Even though the library already provides examples for booleans and streams of booleans, it could also be interesting to add cases for dealing with some three-valued logics, or a case for IEEE1164's std_logic type, used often in VHDL.

Another interesting point is that Coquet defines a `stream` type family, and then proceeds to define several interesting function over `stream`, as well as an instance for the meaning relation. The type family is defined as follows:

```
Definition stream A := nat -> A.
```

If instead of using `nat` in the above definition, we generalized it a bit more (abstracting a new type parameter), we arrive at the concept of an "event stream":

```
Definition events tag A : tag -> A.
```

This is exactly how ForSyDe defines its "signals", so we could model circuits in models of computation other than the synchronous model (which is the one allowed by the current `stream` definition).

5 Results summary

6 Conclusions

References

[1] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. *SIGPLAN Not.*, 34(1):174–184, September 1998.

[2] Thomas Braibant. Coquet: a coq library for verifying hardware. In *Certified Programs and Proofs*, pages 330–345. Springer, 2011.

[3] Shimon Shocken Noam Nisan. *The Elements of Computing Systems: building a modern computer from first principles*. MIT Press, 3rd edition, 2012.

[4] Ingo Sander and Axel Jantsch. Formal system design based on the synchrony hypothesis, functional models, and skeletons. In *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pages 318–323. IEEE, 1999.