Projet PiCoq

Deliverable D5

December 2011

In the year 2011, two case studies have been formalized. First, Thomas Braibant developed a Coq library for verifying hardware and applied it to some simple circuits. His development can be found here: `http://sardes.inrialpes.fr/~braibant/coquet/`.

Second, some results on the higher order calculus HOcore have been formalized. More precisely, Simon Boulier and Alan Schmitt showed in Coq that the Input/Output bisimulation is a correct approximation of barbed congruence and that is it also a decidable relation. The source code of this development is available online at `http://www.irisa.fr/celtique/aschmitt/research/hocore/toc.html`.

Two papers describing these works, "Coquet: a Coq library for verifying hardware" and "Formalisation de HOCore en Coq", are attached to this deliverable.

Coquet: a Coq library for verifying hardware*

Thomas Braibant**

LIG, UMR 5217, INRIA

Abstract. We propose a new library to model and verify hardware circuits in the Coq proof assistant. This library allows one to easily build circuits by following the usual pen-and-paper diagrams. We define a deep-embedding: we use a (dependently typed) data-type that models the architecture of circuits, and a meaning function. We propose tactics that ease the reasoning about the behavior of the circuits, and we demonstrate that our approach is practicable by proving the correctness of various circuits: a text-book divide and conquer adder of parametric size, some higher-order combinators of circuits, and some sequential circuits: a buffer, and a register.

Introduction

Formal methods are widely used in the verification of circuit design, and appear as a necessary alternative to test and simulation techniques. Among them, model checking methods have the advantage of being fully automated but can only deal with circuits of fixed size and suffers from combinatorial explosion. On the other hand, circuits can be formally specified and certified using theorem provers [10,9,14]. For instance, the overall approach introduced in [9,17] to model circuits in higher-order logic is to use predicates of the logic to express the possible behaviour of devices.

We present a study for specifying and verifying circuits in Coq. Our motivations are two-fold. First, there has been a lot of work describing and verifying circuits in logic in the HOL and ACL2 family of theorem provers. However, Coq features dependent types that are more expressive. The Veritas language experiment [10] hinted that these allow for specifications that are both clearer and more concise. We also argue that dependent types are invaluable for developing circuits reliably: some errors can be caught early, when type-checking the circuits or their specifications. Second, most of these works model circuits using a shallow-embedding: circuits are defined as predicates or functions in the logic of the theorem prover, with seldom, if any, way to reason about the devices inside the logic: for instance, functions that operate on circuits must be built at the meta-level [21], which precludes one from proving their correctness. We define a data-type for circuits and a meaning function: we can write (and reason about) Coq functions that operate on the structure of circuits.

* To appear in Proc. CPP, volume 7086 of LNCS, Springer-Verlag, 2011.

** The author has been partially funded by the French projects "Choco", ANR-07-BLAN-0324 and "PiCoq", ANR-10-BLAN-0305.

Circuit diagrams describe the wire connections between gates and have nice algebraic properties [5,15]. While we do not prove algebraic laws, our library features a set of basic blocks and combinators that allows one to describe such diagrams in a hierarchic and modular way. We make precise the interconnection of circuits, yet, we remain high-level because we make implicit the low-level diagram constructs such as wires and ports. Circuit diagrams are also used to present recursive or parametric designs. We use Coq recursive definitions to *generate* circuits of parametric size, e.g., to generate an n-bit adder for a given n. Then, we reason about these functions rather than on the tangible (fixed-size) instantiations of such generators. Circuits modelled by recursion have already been verified in other settings [14,17]. The novelty of our approach is that we derive circuit designs in a systematic manner: we structure circuits generators by mimicking the usual circuit diagrams, using our combinators. Then, the properties of these combinators allow us to prove the circuits correct.

We are interested in two kinds of formal dependability claims. First, we want to capture some properties of well-formedness of the diagrams. Second, we want to be able to express the *functional correctness* of circuits – the fact that a circuit complies to some specification, or that it implements a given function. Obviously, the well-formedness of a circuit is a prerequisite to its functional correctness. We will show that using dependent types, we can get this kind of verification for free. As an example, the type-system of Coq will preclude the user to make invalid compositions of circuits. Hence, we can focus on what is the intrinsic *meaning* of a circuit, and prove that the meaning of some circuits entails a high-level specification, e.g., some functional program.

Our contributions can be summarized as follows: we propose a new framework to model and verify circuits in Coq that allows us to define circuits in a systematic manner by following the usual diagrams; we provide tactics that allow to reason about circuits; we demonstrate that our approach is practicable on practical examples: text-book n-bit adders, high-level combinators, and sequential circuits.

Outline. In §1, we give a small overview of all the basic concepts underlying our methodology to present how the various pieces fit together. We present the actual definitions we use in §2. Then, in §3 and §4, we demonstrate the feasibility of our approach on some examples. We analyse some benefits of using a deep-embedding in §5. Finally, we compare our study to other related work in §6.

1 Overview of our system

We give a global overview of the basic concepts of our methodology first, before giving a formal Coq definition to these notions in the next section. We take this opportunity to illustrate the use of our system to represent parametrized systems through the example of a simple n-bit adder: it computes an n-bit sum and a 1-bit carry-out from two n-bit inputs and a 1-bit carry-in. The recursive construction scheme of this adder is presented in Fig. 1 (data flows from left to right), using a *full-adder*, i.e., a 1-bit adder, as basic building block.

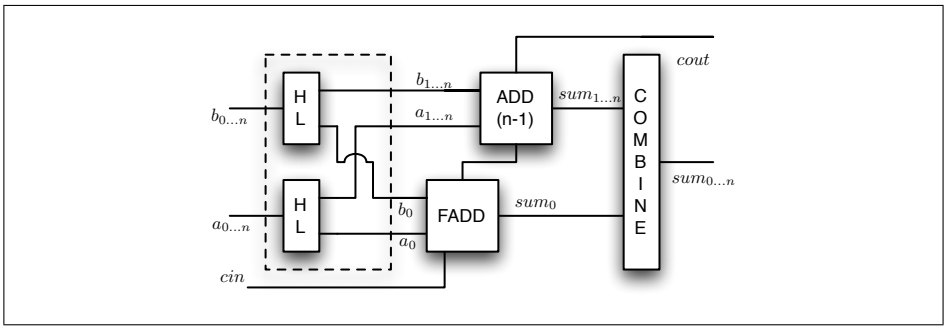


Fig. 1: A recursive n-bit ripple-carry adder

Circuit interfaces. Informally, we want to build circuits that relate n input wires to m output wires, where n, m are integers. For instance, the door `AND` has two inputs and one output. However, using integers to number the wires does not give much structure: how are grouped the $2n + 1$ input wires of the n-bit adder? Hence, we use arbitrary *finite-types* as indexes for the wires rather than integers [11]. A circuit that relates inputs indexed by n to outputs indexed by m has type $\mathbb{C}\ n\ m$, where n and m are types. For instance, the full-adder, a circuit with three inputs and one output, has type $\mathbb{C}\ (\mathbf{1} \oplus (\mathbf{1} \oplus \mathbf{1}))\ (\mathbf{1} \oplus \mathbf{1})$, where $\oplus$ is the disjoint sum (associative to the left) and $\mathbf{1}$ is a singleton type. Hence, the n-bit adder has type $\mathbb{C}\ (\mathbf{1} \oplus \texttt{sumn 1 n} \oplus \texttt{sumn 1 n})\ (\texttt{sumn 1 n} \oplus \mathbf{1})$, where `sumn` $A\ n$ is a n-ary disjoint sum.

Circuits combinators. The n-bit adder is made of several sub-components that are composed together. We use *circuit combinators* (or combining forms [19]) to specify the connection layout of circuits. For instance, in Fig. 1, the dashed-box is built by composing in parallel two `HL` circuits that are then composed serially with a combinator that reorders the wires. These combinators leave implicit the connection points in the circuits and focus on how information flow through the circuit: the wire names given in Fig. 1 do not correspond to variables, and are provided for the sake of readability.

In our "nameless" setting, wires have to be forked and reordered using *plugs*: a plug is a circuit of type $\mathbb{C}\ n\ m$, defined using a map from m to n that defines how to connect an output wire (indexed by m) to an input wire (indexed by n). Since we use functions rather than relations, this definition naturally forbids short-circuits (two input wires connected to the same output wire).

Meaning of a circuit. We now depart from the syntactic definitions of circuits to give an overview of their semantics. We assume a type $\mathbb{T}$ of what is carried by the wires, for instance booleans ($\mathbb{B}$) or streams of booleans ($\texttt{nat} \to \mathbb{B}$). Let x be a circuit of type $\mathbb{C}\ n\ m$. The *inputs* (resp. *outputs*) of x are a finite function *ins* of type $n \to \mathbb{T}$ (resp. *outs* of type $m \to \mathbb{T}$). The *meaning* of x is a relation $x \vdash^n_m \mathit{ins} \bowtie \mathit{outs}$ between *ins* and *outs* that we define by induction on x.

This is an abstract mathematical characterization, which may or may not be computational (we will come back to this point later).

Abstracting from the implementation. The meaning of a circuit is defined by induction on its structure: this relation may be complex and may give information about the internal implementation of a circuit. Thus, we want to move from the definition of this relation, for instance, to give high-level specifications, or to abstract their behavior. These abstractions can be expressed through the following kind of entailment [17]:

$$\forall ins, \forall outs, \texttt{RIPPLE n} \vdash ins \bowtie outs \to R\ outs\ ins$$

We use *data-abstraction* [17] to be more elegant. Indeed, a value of type $\mathbf{1} \oplus \texttt{sumn 1 n} \oplus \texttt{sumn 1 n} \to \mathbb{B}$ is isomorphic to a value of type $\mathbb{B} \times \mathbb{W}_n \times \mathbb{W}_n$ (where $\mathbb{W}_n$ is the type of integers from 0 to 2^n). We use *type-isomorphisms* to give tractable specifications for circuits: we prove that the parametric n-bit adder depicted in Fig. 1 implements the addition with carry function on $\mathbb{W}_n$.

2 Formal development

We now turn to define formally the concepts that were overviewed in the previous section. We use Coq type-classes to structure and parametrize our development.

2.1 Circuit interfaces

We use arbitrary finite types (types with finitely many elements) as interfaces for the circuits, i.e., as indexes for the wires. One can create such finite types by using the disjoint-sum operator $\oplus$ and the one-element type $\mathbf{1}$. This construction can be generalized to n-ary disjoint sums written `sumn A n`, for a given `A`. However, using a single singleton type for all wires can be confusing: there is no way to distinguish one $\mathbf{1}$ from another, except by its position in the type (which is frustrating). Hence, we use an infinite family of singleton types $\mathbf{1}_x$ where x is a *tag*. Circuits are parametrised by some tags, which allows the Coq type-system to rule out some ill-formed combinations. This tagging discipline makes it easy to follow circuit diagrams to define circuits in Coq, without much room for mistakes.

```
Inductive tag (t : string) : Type := _tag : tag t. (** we write 1_t for tag t*)
```

Finite types are defined as a class `Fin A` that packages a duplicate-free list of all the elements of the type `A`, defined along the lines of [8].

2.2 Type isomorphisms

We use type-isomorphisms as "lenses" to express the specification of circuits in user-friendly types, without loss of information. In a nutshell, we define in Coq

an isomorphism between two types A and B as a pair of functions $iso : A \to B$ and $uniso : B \to A$ that are proved to be inverse of each other. We use the notation $A \cong B$ for an isomorphism between A and B, and we define some notations for operations (or instances) that allow us to build such isomorphisms in Fig. 2. The most important instance states the duality between disjoint-sums in the domain of the finite functions to a cartesian product.

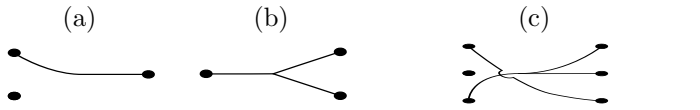
```
Class Iso (A B : Type) :={
 iso : A → B;
 uniso : B → A}.
```

```
Class Iso_Props {A B: Type} (I : Iso A B):= {
 iso_uniso : ∀ (x : B), iso (uniso x) = x;
 uniso_iso : ∀ (x : A), uniso (iso x) = x}.
```

2.3 Plugs

Rewiring circuits of type $\mathbb{C}\ n\ m$ are defined by mapping output wires indexed by m to input wires indexed by n. We define plugs using usual Coq functions to get small and computational definition of maps. (Note that, since we map the indexes of the wires, there is no way to embed an arbitrary function inside our circuits to compute, e.g., the addition of the value carried by two input wires.)

We give three examples: (a) is a circuit that forgets its first input (types must be read bottom-up on diagrams); (b) is a circuit that duplicates its inputs; (c) implements some re-ordering and duplication of the wires. (We leave implicit the associativity of wires on the diagrams.)



$\mathbb{C}\ (n \oplus m)\ m$ $\quad$ $\mathbb{C}\ n\ (n \oplus n)$ $\quad$ $\mathbb{C}\ (n \oplus m \oplus p)\ (p \oplus (n \oplus n))$

A possible implementation for (a) is `fun x ⇒ inr x` and (b) can be implemented as `fun x ⇒ match x with inl e ⇒ e | inr e ⇒ e end`. If the type of the circuit gives enough information, like the examples above, it is possible to define such plugs using proof-search. Indeed, plugs that deal with the associativity of the wires, or even re-orderings, are completely defined by their type, and we use tactics to write the map between wires (it amounts to some case splitting and little automation). Hence, in the formal definition of circuits, we omit the plugs that deal with associativity or re-orderings of the wires, not only for the sake of readability, but also because we do so in the actual Coq code: we leave holes in the code (thanks to the Coq Program feature) that will be filled automatically.

$$\cdot \bullet \cdot \ \frac{A \to \mathbb{T} \cong \sigma \qquad B \to \mathbb{T} \cong \tau}{A \oplus B \to \mathbb{T} \cong (\sigma \times \tau)} \qquad \iota_{\mathtt{x}}\ \frac{}{\mathbf{1}_x \to \mathbb{T} \cong \mathbb{T}} \qquad \frac{A \to \mathbb{T} \cong \sigma}{\mathtt{sumn}\ A\ n \to \mathbb{T} \cong \mathtt{vector}\ \sigma\ \mathtt{n}}$$

Fig. 2: Isomorphisms between types

```
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) (m ⊕ p) → ℂ n m.
```

Fig. 3: Syntax

2.4 Abstract syntax

In the following, we use some basic gates from which all other circuits are defined. Hence, we parametrize the definition of circuits by the type of the gates:

```
Class Techno := techno : Type → Type → Type.
```

Fig. 3 presents the dependent type that models circuits, as defined in Coq. This abstract syntax is strongly typed: it ensures that circuits built using the provided combinators are well-formed: dimensions have to agree, and it is not possible to connect circuits in the wrong direction. (Note that this is not anecdotal: if we were to describe circuits with ports and wires, ensuring these properties would require some boilerplate.) We denote serial composition (`Ser`) with the infix $\triangleright$ symbol, and parallel composition (`Par`) with &. (Note that these definitions do not deal with what transits in the wires.)

2.5 Structural specifications

Let $\mathbb{T}$ be the type of what is carried in the wires. We now define the meaning relation for circuits. For a given circuit of type $\mathbb{C}\ n\ m$, we build a relation between two functions of type $n \to \mathbb{T}$ and $m \to \mathbb{T}$. We define several operations on such functions, in order to express the meaning relation in a legible manner:

```
Context {𝕋 : Type}.
Definition left  {n} {m} (x : (n ⊕ m) → 𝕋) : n → 𝕋 := fun e ⇒ (x (inl _ e)).
Definition right {n} {m} (x : (n ⊕ m) → 𝕋) : m → 𝕋 := fun e ⇒ (x (inr _ e)).
Definition lift {n} {m} (f : m → n) (x : n → 𝕋) : m → 𝕋 := fun e ⇒ x (f e).
Definition app {n m} (x : n → 𝕋) (y : m → 𝕋) : n ⊕ m → 𝕋 :=
 fun e ⇒ match e with inl e ⇒ x e | inr e ⇒ y e end.
```

We define the semantics of a given set of basic gates `tech`: `Techno` by defining instances of the following type-class, (typically, one instance for the boolean setting, and one instance for the boolean stream setting):

```
Class Technology_spec (tech : Techno) 𝕋 :=
   spec : ∀ {a b: Type}, tech a b  → (a → 𝕋)  → (b → 𝕋) → Prop.
```

The meaning relation for circuits is generated by this parameter and rules for each combinator. These rules are presented on Fig. 4 using inference rules rather than the corresponding Coq inductive, for the sake of readability.

2.6 Abstractions

The meaning relation defines precisely the behavior of a circuit, but cannot be used as it is. First, it may be too precise, e.g., by leaking some internal details or imposing constraints between the inputs and the outputs of a circuit that are not relevant from an external point of view. Second, it defines a constraint between the inputs and outputs of a circuit as a relation between two functions $n \to \mathbb{T}$ and $m \to \mathbb{T}$, which is not user-friendly. In his book [17], Melham defines two kinds of abstractions that are relevant here: behavioral abstraction (expressed through the logical entailment of a weak specification R by the meaning relation) and data-abstraction (when the specification is expressed in terms of higher-level types than the above function types).

We combine these two notions to specify that a given circuit realises a specification R up to two type isomorphisms, and to get more concise specifications, we also define the fact that a circuit implements a function `f` up to isomorphisms:

```
Context {n m N M : Type} (Rn : (n→ 𝕋) ≅ N) (Rm : (m→ 𝕋) ≅ M).
Class Realise (c : ℂ n m) (R : N → M → Prop) := realise: ∀ ins outs,
    c ⊢ₘⁿ ins ⋈ outs → R (iso ins) (iso outs).
Class Implement (c : ℂ n m) (f : N → M) := implement: ∀ ins outs,
    c ⊢ₘⁿ ins ⋈ outs → iso outs = f (iso ins).
```

2.7 Atoms and modular proofs

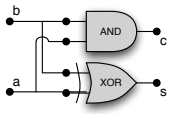
We develop circuits in a modular way: to build a complex circuit, we define a functor that takes as an argument a module that packages the implementations of the sub-components, and the proofs that they meet some specification. This means that our proofs are hierarchical: we do not inspect the definition of the sub-components when we prove a circuit. These functors can then be applied to a module that contains a set of basic doors (of type `Techno`) and its meaning relation (of type `Technology_spec`).

$$\textsc{KSer}\ \frac{x \vdash_m^n ins \bowtie middle \qquad y \vdash_p^m middle \bowtie outs}{x \rhd y \vdash_p^n ins \bowtie outs}$$

$$\textsc{KPar}\ \frac{x \vdash_p^n \texttt{left}\ ins \bowtie \texttt{left}\ outs \qquad y \vdash_q^m \texttt{right}\ ins \bowtie \texttt{right}\ outs}{x \& y \vdash_{p\oplus q}^{n\oplus m} ins \bowtie outs}$$

$$\textsc{KPlug}\ \frac{}{\texttt{Plug}\ f \vdash_m^n ins \bowtie \texttt{lift}\ f\ ins} \qquad \textsc{KLoop}\ \frac{x \vdash_{m\oplus p}^{n\oplus p} \texttt{app}\ ins\ r \bowtie \texttt{app}\ outs\ r}{\texttt{Loop}\ x \vdash_m^n ins \bowtie outs}$$

Fig. 4: Meaning of circuits (omitting the rule for Atom)




```
Context a b s c : string. (*section variables*)
Definition HADD : ℂ (1_a ⊕ 1_b) (1_s ⊕ 1_c) :=
Fork2 (1_a ⊕ 1_b) ▷ (XOR a b s  & AND a b c).
```

Fig. 5: Definition of a half-adder

3 Proving some combinatorial circuits

In this section, we focus on acyclic combinational circuits, and implement some arithmetic circuits. We assume a set of basic gates (`AND`, `XOR` among others, that can all be defined and proved correct starting from `NOR` only). Wires carry booleans, i.e., the meaning relation is defined on booleans for the basic gates. We first illustrate our proof methodology on a half-adder. Then, we present operations on n-bits integers that will be used to specify n-bit adders.

3.1 Proving a half-adder

A half-adder adds two 1-bit binary numbers together, producing a 1-bit number and a carry-out. However, they cannot be chained together since they have no carry in-input. We present a diagram of this circuit, along with its formal definition, in Fig. 5. The left-hand side of the following Coq excerpt is the statement we prove: the circuit `HADD` implements the function `hadd` on booleans (defined as λ(a,b).(a $\oplus$ b, a $\wedge$ b), where $\oplus$ is the boolean exclusive-or, and $\wedge$ is the boolean and) up to isomorphisms (we use the notations from Fig. 2 for isos). The Coq system ask us to give evidence of the right-hand side.

```
Instance HADD_Spec : Implement
(ι_a • ι_b) (* iso on inputs *)
(ι_s • ι_c) (* iso on outputs *)
HADD hadd.
```

```
I : 1_a ⊕ 1_b → 𝔹, O : 1_s ⊕ 1_c → 𝔹
H : HADD ⊢ 1_a⊕1_b / 1_s⊕1_c ins ⋈ outs
=====================
@iso (ι_s • ι_c) O = hadd (@iso (ι_a • ι_b) I)
```

We have developed several tactics that help to prove these kind of goals. First, we automatically invert the derivation of the meaning relation in the hypothesis `H`, following the structure of the circuit, to get rid of parallel and serial combinators. This leaves the user with one meaning relation hypothesis per sub-component in the circuit (plugs included). Second, we use the type-class `Implement` as a dictionary of interesting properties. We use it to make fast-forward reasoning by applying `implements` in any hypothesis stating a meaning relation for a sub-component. The type-class resolution mechanism will look for an instance of `Implement` for this sub-component, and transform the "meaning relation" hypothesis into an equation. (Note that at this point, the user may have to interact with the proof-assistant, e.g., to choose other `Implement` instances than the ones that are picked automatically, but in many cases, this step is automatic.) At this point, the goal looks like the left-hand side of the following excerpt:

```
I : 1_a ⊕ 1_b → 𝔹, O : 1_s ⊕ 1_c → 𝔹
M : (1_a ⊕ 1_b) ⊕ (1_a ⊕ 1_b) → 𝔹
H0: iso M = (fun x ⇒ (x,x)) (iso I)
H1: iso (left O) = uncurry ⊕ (iso (left M))
H2: iso (right O)= uncurry ∧ (iso (right M))
============================
iso O = hadd (iso I)
```

```
I: 𝔹 * 𝔹, O: 𝔹 * 𝔹,
M : (𝔹 * 𝔹) * (𝔹 * 𝔹),
H0: M = (fun x ⇒ (x,x)) I
H1: fst O = uncurry ⊕ (fst M)
H2: snd O = uncurry ∧ (snd M)
==================
O = hadd I
```

Third, we move to the right-hand side of the excerpt: we massage the goal to make some `iso` commute with the `left`, `right` and `app` operations, in order to generalize the goal w.r.t. the isos. (Note that the user may be required to interact with Coq if different isos are applied to the same term in different equations.) Finally, the proof context deals only with high-level data-types, and functions operating on these. The user may then prove the "interesting" part of the lemma.

3.2 n-bits integers

From now, we use a dependently typed definition of n-bits integers, along the lines of the fixed-size machine integers of [16]. We omit the actual definitions of functions when they can be infered from the type. In the following, we prove that various (recursive) circuits implement the `carry_add` function (that adds two n-bit numbers and a carry).

```
Record word (n:nat) := mk_word {val : Z; range: 0 ≤ val < 2^n}.  (* 𝕎_n *)

Definition repr n (x : Z) : 𝕎_n := ...
Definition high n m (x : 𝕎_(n+m)) : 𝕎_m := ...
Definition low n m (x : 𝕎_(n+m)) : 𝕎_n := ...
Definition combine n m (low : 𝕎_n) (high : 𝕎_m) : 𝕎_(n+m) := ...
Definition carry_add n (x y : 𝕎_n) (b : 𝔹) : 𝕎_n * 𝔹 :=
    let e := val x + val y + (if b then 1 else 0) in (e mod 2^n,2^n ≤ e)

Definition Φ^n_x : Iso (sumn 1_x n → 𝔹) (𝕎_n) := ...
```

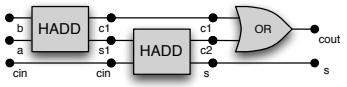
3.3 Two specifications of a 1-bit adder

A full-adder adds two 1-bit binary numbers with a carry-in, producing a 1-bit number and a carry-out, and is built from two half-adders. We present a diagram of this circuit, along with its formal definition in Fig. 6.

From this circuit, we can derive two specifications of interest. First, the meaning of the full-adder can be expressed in terms of a boolean function that mimics the truth-table of the circuit. Second, we can prove that this circuit actually implements the `carry_add` function up-to isomorphism. Both these specifications are proved using the aforementioned tactics, only the interesting parts differ.

```
Instance FADD_1 : Implement
(ι_cin • (ι_a • ι_b)) (* iso on inputs *)
(ι_sum • ι_cout) (* iso on outputs *)
FADD (fun (c,(x,y)) ⇒
  (x ⊕ (y ⊕ c),(x ∧ y) ∨ c ∧ (x ⊕ y))).
```

```
Instance FADD_2 : Implement
(ι_cin • (Φ^1_a • Φ^1_b)) (* iso on inputs *)
(Φ^1_sum • ι_cout) (* iso on outputs *)
FADD (fun (c,(x,y)) ⇒
  carry_add 1 x y c).
```




```
Context a b cin sum cout : string.
Program Definition FADD :
 ℂ (𝟭_cin ⊕ (𝟭_a ⊕ 𝟭_b)) (𝟭_sum ⊕ 𝟭_cout) :=
  (ONE 𝟭_cin & HADD a b "s1" "c1")
  ▷ ... (* associativity plug *)
  ▷ (HADD cin "s1" sum "c2" & ONE 𝟭_"c1")
  ▷ ... (* associativity plug *)
  ▷ (ONE 𝟭_sum & OR "c2" "c1" cout).
```

Fig. 6: Definition of a full-adder

```
Program Fixpoint RIPPLE cin a b cout sum n :
  ℂ (𝟭_cin ⊕ sumn 𝟭_a n ⊕ sumn 𝟭_b n) (sumn 𝟭_sum n ⊕ 𝟭_cout) :=
match n with
  | O ⇒ ... (* Associativity *)
  | S p ⇒ ... ▷ (ONE (𝟭_cin) & HIGHLOWS a b 1 p)
     ▷ ... ▷ (FADD a b cin sum "c" & ONE (sumn 𝟭_a p ⊕ sumn 𝟭_b p))
     ▷ ... ▷ (ONE (sumn 𝟭_sum 1) & RIPPLE "c" a b cout s p)
     ▷ ... ▷ COMBINE sum 1 p & ONE (𝟭_cout)
end.
```

Fig. 7: Implementation of the ripple-carry-adder from Fig. 1

3.4 Ripple-carry adder

We present in Fig. 7 the formal definition of the ripple-carry adder from Fig. 1 (we omit the rewiring plugs). This definition is based on two new circuits to split wires, and combine them. Indeed, to build a $1+n$-bit adder, the lowest-order wire of each parameter is connected to a full-adder, while the n high-order wires of each parameter are connected to another ripple-carry adder. Conversely, the wires corresponding to the sum must be combined together. We use two plugs to define the HL and the COMBINE circuits.

```
Definition HL x n p : ℂ (sumn 𝟭_x (n + p)) (sumn 𝟭_x n ⊕ sumn 𝟭_x p):= Plug ...
Definition COMBINE x n p : ℂ (sumn 𝟭_x n ⊕ sumn 𝟭_x p) (sumn 𝟭_x (n + p)):= Plug ...
```

Then, we prove that these functions on wires implement their counterparts on words. These gates are then easily combined two-by-two to build HIGHLOWS and COMBINES that work with two sets of wires at the same time to get more economical designs (i.e., designs with fewer sub-components).

```
Lemma HL_Spec x n p: Implement
(Φ_x^{n+p}) (Φ_x^n • Φ_x^p) (HL x n p)
(fun x ⇒ (low n p x, high n p x)).
```

```
Lemma COMBINE_Spec x n p: Implement
(Φ_x^n • Φ_x^p) (Φ_x^{n+p}) (COMBINE x n p)
(fun x ⇒ (combine n p (fst x) (snd x))).
```

Finally, we prove by induction on the size of the circuit that it implements the high-level carry_add addition function on words. (Note that this is a high-level

specification of the circuit: the `carry_add` function is not recursive and discloses nothing of the internal implementation of the device.) This boils down to the proof of lemma `add_parts`.

```
Lemma add_parts n m (xH yH: word m) (xL yL : word n) cin:
  let (sumL,middle) := carry_add n xL yL cin in
  let (sumH,cout) := carry_add m xH yH middle in
  let sum := combine n m sumL sumH in
  carry_add (n + m) (combine n m xL xH)(combine n m yL yH) cin = (sum,cout).

Instance RIPPLE_Spec cin a b cout sum n : Implement (RIPPLE cin a b cout s n)
  (ι_cin • (Φ^n_a • Φ^n_b))  (Φ^n_sum • ι_cout)  (fun (c,(x,y)) ⇒ carry_add c x y).
```

This design is simple (a linear chain of 1-bit adders) and slow (each full-adder must wait for the carry-in bit from the previous full-adders). In the next subsection, we address the case of a more efficient adder, which is incidentally more complicated, and a better benchmark for our methodology.

3.5 Divide and conquer adder

A text-book [1] solution to improve on the delay of the previous ripple-carry adder is to use a divide and conquer scheme, and to compute both the sum when there is a carry-in, and the sum when there is no carry-in. It is then possible to compute at the same time the sum for the high-order bits, and the sum for the low order bits. Hence, we build a circuit that computes four pieces of data: s (resp. t), the n-bit sum of the inputs, assuming that there is no carry-in (resp. assuming that there is a carry-in); p the *carry-propagate* bit (resp. g the *carry-generate* bit), which is true when there is a carry-out of the circuit, assuming that there is a carry-in (resp. that there is no carry-in).

We provide a diagram in Fig. 8 that depicts the base case and the recursive case, but we omit the actual Coq implementation for the sake of readability. We prove that this circuit implements the following Coq function:

```
Definition dc n :𝕎_{2^n} * 𝕎_{2^n}  → 𝔹 * 𝔹 * 𝕎_{2^n} * 𝕎_{2^n} := fun (x,y) ⇒
let (s,g) := carry_add 2^n x y false in
let (t,p) := carry_add 2^n x y true in (g,p,s,t).
```

Again, this is a high-level specification w.r.t. the definition of the circuit: it does not disclose how the circuit compute its results (for instance, the `dc` function is not recursive). In a nutshell, the circuit computes in parallel the 4-uple of results for the high-order and low-order part of the inputs. Then, the propagate and generate bits for both parts can be combined by the `PG` circuit to compute the propagate and generate bits for the entire circuit. In parallel, the `FIX` circuit is made of two 2^{n-1}-bit multiplexers (easily defined with a fixpoint using 1-bit multiplexers), and updates the high-order parts of the sum, w.r.t. the propagate and generate carry-bits of the low-order adder.

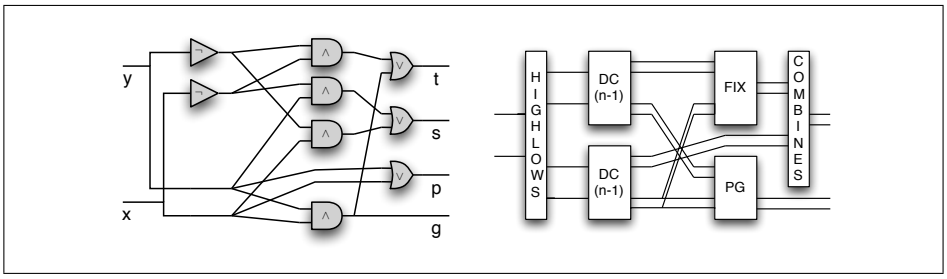


Fig. 8: Divide and conquer adder

4 Sequential circuits: time and loops

While we have focused our case studies on combinational circuits, our methodology can be applied to sequential circuits, with or without the loops that were allowed in the syntax of circuits in §2.4. In this section, the wires carry streams of booleans (of type `nat` $\to \mathbb{B}$), and we assume a basic gate `DFF` that implements the `pre` function (in the particular case of booleans):

```
Definition pre {A} (d : A):
stream A → stream A := fun f t ⇒
match t with | 0 ⇒ d | S p ⇒ f p end.
```

```
Instance DFF_Realise_stream {a out}:
Implement (DFF a out) (ι_a) (ι_out)
(pre false).
```

A buffer. A `DFF` delays one wire by one unit of time; a `FIFO` buffer generalizes this behavior in two dimensions, by chaining layers of `DFF` one after another. This circuit is simple, but is a good example for the use of high-level combinators.

These combinators capture the underlying regularity in some common circuit pattern, for instance replicating a sub-component in a serial or parallel manner.

```
Variable CELL : ℂ n n.
Fixpoint COMPOSEN k : ℂ n n :=
match k with
| 0 ⇒ Plug id
| S p ⇒ CELL ▷ (COMPOSEN p)
end.
```

```
Variable CELL : ℂ n m.
Fixpoint MAP k : ℂ (sumn n k) (sumn m k):=
match k with
| 0 ⇒ Plug id
| S p ⇒ CELL & (MAP p)
end.
```

We prove that the `COMPOSEN` combinator implements a higher-order iteration function, up-to isomorphism: if `CELL` implements a given function `f`, then `COMPOSEN k` implements the iteration of `f`. Respectively, we prove that the `MAP` circuit implements the higher-order `map` function on vectors. Hence, we define a `FIFO` buffer in one-line, and we prove that it implements the function below.

```
Definition FIFO x n k : ℂ (sumn 1_x k) (sumn 1_x k) := COMPOSEN (MAP (DFF x x) k) n.
Definition fifo n k (v : stream (vector 𝔹 k)) : stream (vector 𝔹 k) :=
  fun t ⇒ if n < t then v (t − n) else Vector.repeat k false.
Remark useful_iso : sumn 1 n → stream 𝔹 ≅ stream (vector 𝔹 n) := ...
```

The proof of this specification relies on the above useful isomorphism between groups of wires that carries streams of booleans, and a stream of vectors of

```
Context a load out : string.
Program Definition REGISTER:
ℂ (1_load ⊕ 1_a) 1_out :=
@Loop (1_load ⊕ 1_a) 1_out 1_out
(... ▷ MUX2 a out load "in_dff"
▷ DFF "in_dff" out ▷ Fork2 1_out).
```

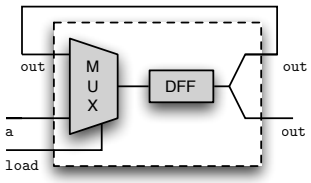


Fig. 9: A memory element

booleans. The proof that the circuit implements a function on streams is done in the same fashion as the proofs from the previous section.

A memory element. Our next goal is to demonstrate how we deal with state-holding structures. Hence, we turn to the implementation of a 1-bit memory element, as implemented in Fig. 9. The register is meant to hold 1-bit of information through time, which does not fit nicely in the `Implement` framework (we cannot easily express the meaning of the register in terms of a stream transformer). Hence, we use a relational specification through the use of `Realise`:

```
Instance Register_Spec : Realise
(... :  1_load ⊕ 1_a → stream 𝔹 ≅ stream 𝔹 ∗ 𝔹) (ι_out) REGISTER
(fun (ins : stream (𝔹 ∗ 𝔹)) (outs : stream 𝔹) ⇒
  outs = pre false (fun t ⇒ if  fst (ins t) then snd (ins t) else outs t)).
```

Here, the state of the register is stored inside the history of the stream (the previous values that were taken by the output). While we do not advocate that this is the nicest way to reason about state holding devices, we were able to prove this specification in the same fashion as the previous combinatorial devices. We leave a more thorough investigation of state-holding devices to future work.

5 Interesting corollaries

We now turn on to investigating some interesting consequences of the use of a concrete data-type to represent circuits. First, we prove that the behavior of combinatorial circuits without delay can be lifted to the stream setting. Second, we build some functions (or interpretations [2]) that operates on circuits.

Lifting combinatorial circuits. The meaning relation is parametrized by the semantics of the basic gates. This can be put to good use to prove the functional correctness of some designs in the boolean setting, and then, to mechanically lift this proof of functional correction to the boolean stream setting (for the same set of gates). For instance, if a loop-less and delay-less circuit implements a function `f` in the boolean setting, we can prove that the very same circuit implements the function `Stream.map f` in the boolean stream setting.

Simulating and checking designs. One feature of our first-order encoding of circuits in Coq is that designs can be checked by simulation before attempting to prove them. This verification is done on the definition that will be proved later, allowing a seamless approach, and remains a valuable help to avoid dead-ends even if we cannot simulate circuit parametrized by a size. We define a simulation function `sim` that works on loop-free circuits, if the user provides a computational interpretation of each basic gate. For instance, we can simulate the adders of §3.

Delay and pretty-printing. Using the same ideas, we can build functions that compute the list of gates of circuits (with or without loops), or compute the length of the critical path in combinatorial circuits. While this is more anecdotal, and less directly useful than the previous simulation function, these functions are still interesting: one could, for instance, prove that some complex designs meet some time (or gate-count) complexity properties. (Note that is the only place where we exploit the finiteness of types.)

6 Comparisons with related work

Verifying circuits with theorem provers. There has been a substantial amount of work on specification and verification of hardware in HOL. In [9,17], HOL is used as a hardware description language and as a formalism to prove that a design meets its specification. They model circuits as predicates in the logic, using a shallow-embedding that merges the architecture of a circuit and its behavior. Building on the former methodology, [21] defines a compiler from a synthetisable subset of HOL that creates correct-by-construction clocked synchronous hardware implementations of mathematical functions specified in HOL. This methodology allows the designer to focus on high-level abstraction instead of reasoning and verifying at the gate level, admitting the existence of some base low-level circuits (like the addition on words [13]). By contrast, our work complements their behavioral "correct by design" synthesis from a subset of the high-level language of the theorem-prover with structural verification of circuits.

In the Boyer-Moore theorem prover (untyped, quantifier-free and first-order), Brock and Hunt proved the correctness of functions that generate correct hardware designs. They studied the correctness of an arithmetic and logic unit, parametrised by a size [14]. This verified synthesis approach was used to verify a microprocessor design [4]. While our proofs are not as automated, and our examples are less ambitious, we are able to prove higher-order circuits. Moreover, the dependent-types we use are helpful when defining complex circuits.

In Coq, Paulin-Mohring [18] proved the correctness of a multiplier unit, using a shallow-embedding similar to the methodology used in HOL: circuits are modelled as functions of the Coq language. More recently, [6] investigated how to take advantage of dependent types and co-inductive types in hardware verification: they use a shallow embedding of Mealy automata to describe sequential circuits. By contrast with both works, we use a deep-embedding of circuits in

Coq, that makes explicit the definition of circuits. We still need to investigate the examples of sequential circuits studied in these papers.

Algebraic definitions of circuits. Circuit diagrams have nice algebraic properties. Lafont [15] studied the algebraic theory of boolean circuits and Hinze [12] studied the algebra of parallel prefix circuits. Both settings are close to ours: however the former focused on the algebraic structure of circuits, while the latter defined combinators that allows one to model (and prove correct using algebraic reasoning) all standard designs of a restricted class of circuits.

Functional languages in hardware design. Sheeran [20] made a thorough review of the use of functional languages in hardware design, and of the challenges to address. Our work is a step towards one of them: the design and verification of parametrized designs, through the use of circuit combinators. Lava [2] is a language embedded in Haskell to describe circuits, allowing one to define parametric circuits or higher-order combinators. While much of our goals are common, one key difference is that our encoding of circuits in Coq avoids the use of bound variables (we use only combinators). Moreover, we use dependent types, that are required to deal precisely with parametric circuits. Finally, we prove the correctness of these parametric circuits in Coq, while verification in Lava is reduced to the verification of finite-size circuits.

7 Conclusion and future works

We have presented a deep-embeding of circuits in the Coq proof-assistant that allows to build and reason about circuits, proving high-level specifications through the use of type-isomorphism. We have demonstrated that dependent types are useful to prove automatically some well-formedness conditions on the circuits, and help to avoid time consuming mistakes. Then, we proved by induction the correctness of some arithmetic circuits of parametric size: this could not have been possible without mimicking the structure of the usual circuit diagrams to define circuit generators in Coq. The formal development accompanying this paper is available from the author's web-page [3].

In the immediate future, we plan to continue the case studies described in §3. In particular, we would like to investigate how to construct parallel prefix circuits in our framework [12,20], and to investigate combinational multipliers. In the more distant future, it would be interesting to study some front-ends to automatically generate some circuits: this could range to the reduction of the boilerplate inherent to the definition of plugs, to the compilation of circuits from automaton. A major inspiration on behavioral synthesis is the work of Ghica [7]. We also look forward to studying how our methodology applies to other settings than booleans or streams of booleans. For instance, if we move from booleans to the three-valued Scott's domain (unknown, true, false), we may interpret circuits in the so-called constructive semantics. We also hope that some of our methods could be applied to the probabilistic setting.

Acknowledgements. We thank D. Pous for his precious supervision; the reviewers who made useful comments; J. Alglave, P. Boutillier, J. Planul, M. Hague who commented a draft; and J. Vuillemin for his encouragements.

References

1. A. V. Aho and J. D. Ullman. *Foundations of Computer Science*. Computer Science Press, W. H. Freeman and Company, 1992.
2. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *Proc. ICFP*, pages 174–184. ACM Press, 1998.
3. T. Braibant. `http://sardes.inrialpes.fr/~braibant/coquet`, June 2011.
4. B. Brock and W. A. Hunt Jr. The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor. *Formal Methods in System Design*, 11(1):71–104, 1997.
5. C. Brown and G. Hutton. Categories, allegories and circuit design. In *Proc. LICS*, pages 372–381. IEEE Computer Society, 1994.
6. S. Coupet-Grimal and L. Jakubiec. Certifying circuits in type theory. *Formal Asp. Comput.*, 16(4):352–373, 2004.
7. D. R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In *Proc. POPL*, pages 363–375, 2007.
8. G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.
9. M. Gordon. Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware. Technical Report UCAM-CL-TR-77, Cambridge Univ., Computer Lab, 1985.
10. F. K. Hanna, N. Daeche, and M. Longley. Veritas$^+$: A Specification Language Based on Type Theory. In *Hardware Specification, Verification and Synthesis*, LNCS, pages 358–379. Springer, 1989.
11. J. R. Harrison. A HOL theory of Euclidean space. In *Proc. TPHOLs*, volume 3603 of *LNCS*, pages 114–129. Springer, 2005.
12. R. Hinze. An Algebra of Scans. In *MPC*, LNCS, pages 186–210. Springer, 2004.
13. J. Iyoda. Translating HOL functions to hardware. Technical Report UCAM-CL-TR-682, Cambridge Univ., Computer Lab, April 2007.
14. W. A. Hunt Jr. and B. Brock. The Verification of a Bit-slice ALU. In *Hardware Specification, Verification and Synthesis*, volume 408 of *LNCS*, pages 282–306. Springer, 1989.
15. Y. Lafont. Towards an algebraic theory of boolean circuits. *Journal of Pure and Applied Algebra*, 184:257–310, 2003.
16. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
17. T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.
18. C. Paulin-Mohring. Circuits as Streams in Coq: Verification of a Sequential Multiplier. In *TYPES*, pages 216–230, 1995.
19. M. Sheeran. μFP, A Language for VLSI Design. In *LISP and Functional Programming*, pages 104–112, 1984.
20. M. Sheeran. Hardware Design and Functional Programming: a Perfect Match. *J. UCS*, 11(7):1135–1158, 2005.
21. K. Slind, S. Owens, J. Iyoda, and M. Gordon. Proof producing synthesis of arithmetic and cryptographic hardware. *Formal Asp. Comput.*, 19(3):343–362, 2007.

Formalisation de HOCore en Coq

Simon Boulier[1] & Alan Schmitt[2]

1: ENS Cachan - Antenne de Bretagne
simon.boulier@gmail.com
2: INRIA
alan.schmitt@inria.fr

Résumé

Nous présentons les premiers résultats de la formalisation de propriétés du calcul de processus d'ordre supérieur HOCore [8] dans l'assistant de preuve Coq. Nous décrivons notre choix de représentation des lieurs de HOCore, nous basant sur l'approche canonique de Pollack et al [14]. Nous donnons la représentation de différentes notions de bissimulations, puis la preuve formelle de la correction de l'IO-bissimilarité par rapport à l'équivalence contextuelle barbue, correspondant à un des théorèmes fondamentaux de [8]. Nous montrons également que l'IO-bissimilarité est décidable. L'objectif de ce travail est de montrer l'utilité de Coq et de la représentation canonique pour prouver des propriétés de calculs d'ordre supérieur.

1. Introduction

Les calculs de processus ont été initialement développés pour modéliser des systèmes parallèles et communicants. Ces calculs ont ensuite été étendus pour prendre en compte la migration de systèmes, par exemple en introduisant une notion *d'ordre supérieur* dans le π-calcul [16]. La recherche autour des calculs d'ordre supérieur est très active et porte sur leur expressivité, des définitions possibles pour la notion d'équivalence ou les encodages dans des versions sans ordre supérieur. Les preuves accompagnant ces travaux sont la plupart du temps faites à la main et leur taille ou leur complexité peuvent laisser subsister un doute sur leur correction. Il est ainsi possible d'oublier de considérer un cas, ou de supposer trop rapidement que deux cas sont identiques et de ne pas détailler le deuxième.

L'utilisation d'un assistant de preuves permet d'éviter ces écueils et donne une garantie très forte dans la correction des résultats. De plus, cela impose la reformulation et la simplification du problème afin que celui-ci soit précisément posé. En particulier, dans notre cas comme dans le défi PoplMark [1], il est crucial de formellement définir les notions de lieurs et d'α-conversion.

Nos travaux s'inscrivent dans le cadre du projet ANR PiCoq, [1] qui a pour objectif de développer un environnement permettant la vérification formelle de propriétés de programmes distribués à l'aide de l'assistant de preuve Coq [11]. Plus précisément, nous cherchons à formaliser en Coq certains résultats établis pour le calcul de processus d'ordre supérieur HOCore [8]. HOCore est un calcul minimal, au sens du nombre des opérateurs, mais est suffisamment riche pour être Turing complet et pour fournir un cadre permettant d'étudier des notions d'équivalence. On peut ainsi y définir une bissimilarité très simple, appelée IO-bissimilarité, et montrer qu'elle coïncide avec la congruence barbue.

Nous avons choisi de formaliser ce calcul pour valider certains choix techniques visant principalement la représentation des variables liées et l'α-conversion. Nous voulons ainsi évaluer la facilité d'utilisation de l'approche canonique locale [14] en prouvant des résultats portant sur les bissimulations. Nous nous intéressons en particulier à l'égalité entre IO-bissimilarité et congruence

1. http://sardes.inrialpes.fr/collaborations/PiCoq/

barbue. La première relation est simple à établir entre deux processus : on ne regarde que comment ils interagissent avec leur environnement, en ne prenant pas en compte les communications internes. La deuxième est l'équivalence classique de processus, qui spécifie que l'on ne verra jamais de différence observable entre les deux processus lors de leurs exécutions, quel que soit le contexte dans lequel on les plonge ; ces notions sont formellement définies en sections 2.4 et 2.5. Montrer l'égalité de ces deux relations revient à prouver deux inclusions ; le travail présenté ici porte sur la première : deux processus IO-bissimilaires sont équivalents. Nous montrons également que l'IO-bissimilarité est décidable.

Nous avons rencontré quelques erreurs en formalisant [8], que ce soit au niveau de la définition de bissimulation (problème d'α-équivalence dans la réception de messages) ou dans des esquisses de preuves (argument d'induction ne fonctionnant pas). Ces erreurs ne portent pas préjudice aux résultats, mais nous pensons que notre version formelle propose plusieurs simplifications et éclaircissements qui peuvent aider à la compréhension de l'article initial.

Les contributions de cet article sont les suivantes. Nous proposons une formalisation de HOCore en Coq basée sur l'approche canonique locale des lieurs, nous montrons que l'IO-bissimilarité est correcte par rapport à la congruence barbue et qu'elle est décidable. De plus, nous simplifions la présentation de HOCore et nous essayons d'apporter une intuition sur certaines propriétés de ses propriétés, comme la décidabilité de la congruence barbue.

L'article est structuré comme suit. Dans un premier temps, nous introduisons le calcul HOCore (section 2) en clarifiant sa sémantique et certaines de ses propriétés. Nous détaillons ensuite les modifications apportées pour sa formalisation (section 3). Enfin, nous présentons certains détails du développement Coq (section 4). Les travaux connexes sont abordés en section 5.

Le développement en Coq est accessible à l'adresse suivante : `http://www.irisa.fr/celtique/aschmitt/research/hocore/toc.html`.

2. Présentation de HOCore

2.1. Syntaxe

Le calcul HOCore peut être vu comme une restriction du π-calcul d'ordre supérieur, auquel on aurait enlevé l'opérateur de restriction de nom. Il peut également être vu comme un λ-calcul parallèle, où l'application d'une fonction $\lambda x.P$ à un argument Q est remplacée par la communication sur un canal arbitraire a entre un envoi de message $\overline{a}\langle Q\rangle$ mis en parallèle d'une réception de message $a(x).P$.

La syntaxe de HOCore est la suivante.

$$P ::= a(x).P \;\big|\; \overline{a}\langle P\rangle \;\big|\; P \parallel P \;\big|\; x \;\big|\; \mathbf{0}$$

Un processus P peut soit être une réception de message sur le canal a, notée $a(x).P$, soit une émission de message sur le canal a, notée $\overline{a}\langle P\rangle$, soit la mise en parallèle de processus $P \parallel Q$, soit une variable x, soit le processus inactif $\mathbf{0}$. Nous supposons qu'il existe une infinité dénombrable de noms de canaux et de variables.

Nous détaillons la sémantique du calcul ci-dessous, mais en donnons ici la règle principale : la communication entre émission et réception de message.

$$\overline{a}\langle P\rangle \parallel a(x).Q \longrightarrow [P\,/\,x]Q$$

L'opération $[P/x]Q$ est la *substitution* de la variable x par le processus P dans Q. Nous détaillerons formellement cette notion après avoir abordé les questions de variables et d'α-conversion.

Notons que le calcul est asynchrone : l'émission de message n'a pas de continuation. Dans un calcul synchrone, l'envoi de message est de la forme $\overline{a}\langle P\rangle.Q$, où le processus Q est la continuation démarrée

après l'émission du message sur a. Considérer un calcul synchrone ne change pas les propriétés fondamentales de HOCore, mais rend leurs preuves plus complexes, comme suggéré en section 4.7.

Variables Une variable x est dite *liée* si elle est sous la portée d'un lieur pour cette variable (une réception de message $a(x).P$), *libre* sinon. Par exemple, dans le processus $a(x).(P \parallel y)$ avec $x \neq y$, les occurrences de x dans P sont liées, mais y est libre.

Une variable x est dite *fraîche* par rapport à un terme P (noté $x \sharp P$) si elle n'y apparaît pas. Par exemple, x est fraîche dans $y \parallel y$ ou $a(y).y$ si et seulement si $x \neq y$.

Congruence structurelle Une relation $\mathcal{R}$ est une *congruence* si c'est une relation d'équivalence (réflexive, symétrique, transitive) respectant les différents constructeurs du langage (par exemple si $P \ \mathcal{R} \ Q$ alors nous avons $a(x).P \ \mathcal{R} \ a(x).Q$). La *congruence structurelle*, notée $\equiv$, est la plus petite congruence telle que la composition parallèle soit associative, commutative et d'élément neutre $\mathbf{0}$.

$$P \parallel Q \equiv Q \parallel P \qquad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \qquad P \parallel \mathbf{0} \equiv P$$

2.2. Sémantique

La sémantique de HOCore est définie par un système de transitions étiquetées (LTS). Les états sont les processus et les étiquettes sont de la forme $a(x)$ pour une réception, $\overline{a}\langle P\rangle$ pour une émission et τ pour une communication interne. On étend la notion de variable liée aux étiquettes de la manière suivante : x est liée par $a(x)$, aucune variable n'est liée par $\overline{a}\langle P\rangle$ ou τ. Voici les règles qui définissent le système de transition :

Inp $a(x).P \xrightarrow{a(x)} P$;
Out $\overline{a}\langle P\rangle \xrightarrow{\overline{a}\langle P\rangle} \mathbf{0}$;
Act1 si $P \xrightarrow{\alpha} P'$ alors $P \parallel Q \xrightarrow{\alpha} P' \parallel Q$ si les variables liées de α ne sont pas libres dans Q ;
Tau1 si $P \xrightarrow{\overline{a}\langle P''\rangle} P'$ et $Q \xrightarrow{a(x)} Q'$ alors $P \parallel Q \xrightarrow{\tau} P' \parallel [P'' \,/\, x]Q'$;
Act2 et Tau2 sont les règles symétriques.

La condition sur les règles Act1 et Act2 est requise pour éviter la capture de variables libres : sinon, nous pourrions avoir des transitions de la forme $x \parallel a(x).x \xrightarrow{a(x)} x \parallel x$. Afin de ne pas restreindre les transitions possibles, la sémantique est définie à α-conversion près : on s'autorise le renommage de variables liées à tout moment. Les conséquences d'un tel choix ne sont pas bénignes, nous y reviendrons en section 3.1.

2.3. Expressivité

HOCore est qualifié de minimal, car il ne contient que le strict nécessaire à l'ordre supérieur. Par exemple, il n'inclut pas d'opérateurs de restriction ou de réplication. HOCore est tout de même Turing complet : un encodage fidèle des machines de Minsky est présenté dans [8]. Par conséquent, le problème de la terminaison est indécidable.

2.4. Équivalence de processus

Une des questions cruciales de l'étude de calculs de processus est de savoir si deux processus « font la même chose ». Ainsi, dans l'optique de la programmation modulaire, on doit être capable de dire si deux bibliothèques logicielles sont interchangeables. Deux processus sont équivalents si, dans n'importe quel contexte, ce que l'on observe de leur activité est similaire. Formellement, on définit la *congruence barbue*, $\simeq$, comme la plus grande relation symétrique telle que :

– si $P \simeq Q$ et $P \xrightarrow{\tau} P'$ alors il existe un Q' tel que $Q \xrightarrow{\tau} Q'$ et $P' \simeq Q'$: la congruence barbue est préservée par réductions ;
– si $P \simeq Q$ alors $C[P] \simeq C[Q]$ pour tout contexte C, un contexte étant un processus avec un trou : la congruence barbue est une congruence ;
– si $P \simeq Q$ et $P \xrightarrow{\overline{a}\langle P''\rangle} P'$ alors il existe Q' et Q'' tels que $Q \xrightarrow{\overline{a}\langle Q''\rangle} Q'$: la congruence barbue met en relation des processus avec les mêmes observables, ou barbes, qui sont ici la possibilité d'émettre un message sur un canal donné.

La quantification sur les contextes de la deuxième clause est une des principales causes de la difficulté d'établir l'équivalence de deux processus. On utilise donc généralement d'autres relations, plus faciles à manipuler, qui ne prennent en compte que les interactions des processus avec leur environnement : des bissimulations. Il est par contre nécessaire de montrer que ces relations coïncident avec la congruence barbue.

2.5. Bissimulations

Les *bissimulations* sont des relations d'équivalence coinductives qui mettent en relation des processus ayant des interactions similaires avec l'environnement. Nous ne rappelons ici que les définitions de [8] dont nous avons besoin.

– Une relation $\mathcal{R}$ est dite *input* si dès que $P \mathcal{R} Q$ et $P \xrightarrow{a(x)} P'$, alors il existe Q' tel que $Q \xrightarrow{a(y)} Q'$ et pour tout z frais pour P et Q nous avons $[z \,/\, x]P' \mathcal{R} [z \,/\, y]Q'$.
– Une relation $\mathcal{R}$ est dite *output* si dès que $P \mathcal{R} Q$ et $P \xrightarrow{\overline{a}\langle P''\rangle} P'$, alors il existe Q' et Q'' tels que $Q \xrightarrow{\overline{a}\langle Q''\rangle} Q'$, $P' \mathcal{R} Q'$ et $P'' \mathcal{R} Q''$.
– Une relation $\mathcal{R}$ est dite *variable* si $P \mathcal{R} Q$ et $P \equiv P' \parallel x$ impliquent qu'il existe Q' tel que $Q \equiv Q' \parallel x$ et $P' \mathcal{R} Q'$.
– Une relation $\mathcal{R}$ est une *IO-bissimulation* si elle est symétrique, input, output et variable.
– Une relation $\mathcal{R}$ est une *τ-bissimulation* si $P \mathcal{R} Q$ et $P \xrightarrow{\tau} P'$ impliquent qu'il existe Q' tel que $Q \xrightarrow{\tau} Q'$ et $P' \mathcal{R} Q'$.

La définition d'une relation input dans [8] est la suivante : « $P \mathcal{R} Q$ et $P \xrightarrow{a(x)} P'$ implique qu'il existe Q' tel que $Q \xrightarrow{a(x)} Q'$ et $P' \mathcal{R} Q'$. » Cette définition a pour conséquence que si P peut faire une réception en ayant comme variable liée x, alors Q doit faire une réception *en utilisant la même variable liée*, donc x ne doit pas être libre dans Q. Ce problème est mineur, car HOCore possède la propriété particulière que deux processus équivalents ont les mêmes variables libres. Ainsi, comme x n'est pas libre dans P, il n'est pas libre non plus dans Q. Cette propriété est cependant fausse dès que l'on ajoute un opérateur de restriction de nom. Nous préférons ainsi prendre une définition rigoureuse pour la suite de ce développement.

Un deuxième commentaire au sujet de la relation input porte sur le nombre limité de tests qu'elle exige. En effet, un processus arbitraire pourrait être communiqué par l'environnement, et il n'est pas immédiat que simplement tester des noms frais suffise pour avoir une relation correcte. C'est néanmoins le cas, pour une raison similaire à la correction des bissimulations normales dans HOπ [16] : les seules actions que l'on peut faire après avoir reçu un message sont de le faire suivre, éventuellement inclus dans un autre message, ou de l'exécuter un certain nombre de fois. Il suffit donc de tester, en utilisant un marqueur frais, ici une variable fraîche, que les deux processus ont bien le même comportement, indépendamment du processus reçu.

Notons également que la relation output vérifie que les processus émis sont bissimilaires, alors que la congruence barbue ne le fait pas. Ce n'est pas nécessaire pour cette dernière : elle peut en effet utiliser un contexte capturant le message envoyé pour ensuite le tester. La preuve, plutôt complexe, de cette propriété correspond au lemme 4.15 de [8]. Elle n'est pas détaillée ici car elle fait partie de la preuve de complétude de l'IO-bissimilarité que nous n'avons pas encore formalisée.

Pour simplifier la manipulation de la notion de relation « variable », nous définissons le prédicat $\mathsf{remove}_x(P, P')$ qui est satisfait si et seulement si P' est P dans lequel on a substitué exactement une

occurrence de x par $\mathbf{0}$. On montre facilement que $\mathsf{remove}_x(P, P')$ implique $P \equiv P' \parallel x$. Pour le sens inverse, il faut prendre en compte le fait que P' a pu être réordonné par la congruence structurelle. Nous utilisons désormais cet opérateur pour la définition d'une relation *variable*.

On dit que deux processus P, Q sont *IO-bissimilaires*, noté $P \sim Q$, s'il existe une IO-bissimulation $\mathcal{R}$ telle que $P \mathcal{R} Q$. La relation d'IO-bissimilarité ainsi définie est la plus grande IO-bissimulation au sens de l'inclusion. Cette définition est cohérente, car les critères de bissimulation sont monotones.

Une propriété remarquable de HOCore est que toutes les bissimulations usuelles — bissimulations normale, contextuelle, d'ordre supérieur, synchrone ou asynchrone — sont équivalentes et coïncident avec l'IO-bissimilarité et avec la congruence barbue. De plus, vérifier que deux processus P et Q sont IO-bissimilaires est décidable, ce que l'on peut montrer par induction sur la taille des processus [2] en appliquant l'algorithme suivant (les cas symétriques où P répond à Q sont omis).

- Si P est une composition parallèle de $\mathbf{0}$, on répond positivement si Q est une composition parallèle de $\mathbf{0}$ et négativement sinon. Ce cas de base correspond au cas où aucune des règles « input », « output » ou « variable » ne s'applique.
- Si on a $\mathsf{remove}_x(P, P')$, on teste si P' est bissimilaire à Q' pour tout Q' tel que $\mathsf{remove}_x(Q, Q')$ est vrai. Si une réponse est positive, on répond positivement, sinon on répond négativement. Le processus termine, car seulement un nombre fini de Q' sont testés, et P' et Q' sont plus petits.
- Soit z une variable fraîche pour P et Q. Si $P \xrightarrow{a(x)} P'$, on teste pour tout Q' tel que $Q \xrightarrow{a(y)} Q'$ si $[z \,/\, x]P'$ est bissimilaire à $[z \,/\, x]Q'$. Si une réponse est positive, on répond positivement, sinon on répond négativement. Le processus termine, car un nombre fini de Q' sont testés, et $[z \,/\, y]P'$ et $[z \,/\, y]Q'$ sont plus petits que P et Q respectivement.
- Si $P \xrightarrow{\overline{a}\langle P'' \rangle} P'$, pour tous Q' et Q'' tels que $Q \xrightarrow{\overline{a}\langle Q'' \rangle} Q'$, on teste si P' est bissimilaire à Q' et si P'' est bissimilaire à Q''. Si deux réponses sont simultanément positives, on répond positivement, sinon on répond négativement. Comme précédemment, la procédure termine, car un nombre fini de tests sur des processus plus petits sont générés.

Deux commentaires sur le test « input » doivent être faits. Tout d'abord, nous devrions considérer toutes les variables fraîches possibles. Nous prouvons dans le développement en Coq que l'IO-bissimilarité définie ci-dessus coïncide avec la version où on ne considère qu'une seule variable fraîche z. En d'autres termes, un quantificateur universel est remplacé par un quantificateur existentiel, ce qui est nécessaire pour avoir une procédure décidable. Ensuite, nous prétendons qu'il n'existe qu'un nombre fini de Q' tels que $Q \xrightarrow{a(y)} Q'$. Ce n'est en fait pas le cas à cause de l'α-conversion. Néanmoins, si nous restreignons l'α-conversion à la variable y sur laquelle on fait la réception, on ne génèrera qu'un nombre fini de processus $[z/y]Q'$. Nous verrons dans la prochaine section comment éviter ces problèmes d'α-équivalence.

La combinaison de la décidabilité de l'IO-bissimilarité et du fait qu'elle coïncide avec la congruence barbue implique donc que la congruence barbue est décidable. Ceci peut paraître paradoxal : on peut décider si deux processus font la même chose, mais pas s'ils terminent ! Pour avoir une intuition de ce phénomène, on peut imaginer que l'équivalence considérée est l'égalité syntaxique. Cette équivalence est clairement décidable, et ne présume en rien de l'expressivité du calcul. La situation est similaire pour HOCore : l'équivalence de processus est très fine. On montre dans [8] qu'elle se résume à la congruence structurelle étendue avec la règle suivante, où l'on note $\prod_k P$ la composition parallèle de k copies de P :

$$a(x).\left(P \parallel \prod_{k-1} a(x).P\right) \simeq \prod_{k} a(x).P.$$

En d'autres termes, à congruence structurelle et cette règle près, il n'existe dans HOCore qu'une manière d'exprimer un comportement donné. C'est pour cela que l'on peut facilement décider si deux processus ont le même comportement.

2. Intuitivement, la taille correspond au nombre de variables, d'opérateurs parallèles et de préfixes.

Le but de notre projet de formalisation est de prouver que l'IO-bissimilarité et la congruence barbue coïncident, et que l'IO-bissimilarité est décidable. Nous présentons ici une première partie de ce travail : la formalisation de HOCore en Coq, en prenant soin des problèmes d'α-équivalence, l'inclusion (la correction) de l'IO-bissimilarité par rapport à la congruence barbue et la décidabilité de l'IO-bissimilarité.

3. Modélisation de HOCore en Coq

Pour simplifier et rendre plus formelles les preuves, nous avons adapté trois éléments de HOCore : nous avons utilisé une représentation canonique des termes, introduit l'utilisation d'abstractions locales lors de la réception de messages et ajouté une règle dans le LTS pour la gestion des variables.

3.1. Représentation canonique des termes

Dans la version de HOCore que nous avons présentée, les termes $a(x).x$ et $a(y).y$ (avec $x \neq y$) ne sont pas égaux : il ne diffèrent que par le nom de la variable liée par la réception sur a. On dit qu'ils sont *α-équivalents*. Beaucoup de définitions et de propriétés dépendent de manière directe ou détournée de l'α-équivalence, qui ne peut donc pas être ignorée dans les preuves. Un premier exemple est la définition de notre LTS ci-dessus : le processus $x \parallel a(x).x$ n'a pas de transition, mais le processus α-équivalent $x \parallel a(y).y$ a une transition $a(y)$.

Une approche classique, que ce soit pour des calculs séquentiels tel le λ-calcul ou des calculs de processus, consiste à travailler « modulo α-équivalence » et choisir à la volée des noms ne posant pas de problèmes pour les variables liées, comme avec la « convention de Barendregt » qui suppose que les variables libres et liées aient des noms différents, quitte à renommer les variables liées au moment opportun. Une telle approche n'est malheureusement pas satisfaisante dans le but d'une formalisation dans un assistant de preuve. De plus, la représentation classique des lieurs impose de faire particulièrement attention lors de la définition de la substitution pour éviter la capture de variables.

Plusieurs solutions ont été explorées pour pallier ce problème, comme l'approche nominale, l'utilisation d'indices de De Bruijn ou la différentiation entre les ensembles de variables libres et liées. L'approche nominale consiste à considérer directement les classes d'équivalences engendrées par l'α-conversion [13]. Cette approche est bien intégrée dans l'assistant de preuve Isabelle [7]. Elle permet de raisonner d'une manière très proche des preuves papiers en utilisant une syntaxe concrète classique tout en s'appuyant sur une infrastructure rendant égaux les termes α-équivalents. Cette infrastructure permet également à l'utilisateur de spécifier le contexte dans lequel il se trouve lorsqu'il utilise un principe d'induction et ainsi indiquer les noms à ne pas utiliser comme lieurs. Nominal Isabelle a été utilisé pour modéliser des calculs de processus comme le psi-calcul [4]. Il est possible d'utiliser une approche nominale en Coq, comme illustré par [2], mais comme l'implémentation de celle-ci utilise la technique des lieurs « localement anonyme », nous avons préféré une approche plus directe.

La technique localement anonyme utilise deux espaces de noms différents. Les noms libres sont des noms, alors que les noms liés utilisent des indices de De Bruijn. Utiliser deux espaces de noms simplifie grandement la substitution : les variables libres étant différentes des lieurs, elles ne peuvent pas être capturées durant une substitution. De plus, les indices de De Bruijn garantissent l'unicité de la représentation d'un terme. Un problème est cependant causé par l'utilisation de ces indices : étant donné un lieur, déterminer quelles variables lui correspondent demande un calcul. Cela gêne non seulement la lecture du terme, mais aussi la définition de la substitution, car il faut calculer où substituer le terme. Cette approche est souvent utilisée en Coq, notamment dans le cadre du défi PoplMark [3].

Nous avons décidé pour notre développement de suivre l'approche canonique de Pollack et al [14]. Les noms sont toujours divisés en deux ensembles (noms libres, ou globaux, et noms liés, ou locaux),

mais les lieurs utilisent des noms. Ces noms liés ne sont pas choisis librement, mais calculés en fonction du terme lié. On peut ainsi considérer qu'un terme canonique est un représentant de sa classe d'α-équivalence. Nous verrons ci-après que ces noms ne changent pas lors de la réduction, en particulier parce que HOCore n'autorise pas que l'on calcule sous les lieurs, qui sont les réceptions.

Nous notons $x, y, z, \ldots$ les *variables locales* (lvar) et $X, Y, Z, \ldots$ les *variables globales* (gvar). La syntaxe de notre calcul est désormais la suivante.

$$P ::= a(x).P \;\Big|\; \overline{a}\langle P\rangle \;\Big|\; P \parallel P \;\Big|\; x \;\Big|\; X \;\Big|\; \mathbf{0}$$

La substitution est définie sans se soucier de la capture de variables libres, puisque nous restreindrons les termes de telle sorte qu'ils n'aient pas de variable locale libre.

$$\begin{aligned}
[P\,/\,X]X &= P & [P\,/\,x]x &= P\\
[P\,/\,X]Y &= Y \quad \text{si } X \neq Y & [P\,/\,x]y &= y \quad \text{si } x \neq y\\
[P\,/\,X]x &= x & [P\,/\,x]X &= X\\
[P\,/\,X]\mathbf{0} &= \mathbf{0} & [P\,/\,x]\mathbf{0} &= \mathbf{0}\\
[P\,/\,X](Q \parallel R) &= [P\,/\,X]Q \parallel [P\,/\,X]R & [P\,/\,x](Q \parallel R) &= [P\,/\,x]Q \parallel [P\,/\,x]R\\
[P\,/\,X]\overline{a}\langle Q\rangle &= \overline{a}\langle [P\,/\,X]Q\rangle & [P\,/\,x]\overline{a}\langle Q\rangle &= \overline{a}\langle [P\,/\,x]Q\rangle\\
[P\,/\,X]a(y).Q &= a(y).[P\,/\,X]Q & [P\,/\,x]a(y).Q &= a(y).[P\,/\,x]Q \quad \text{si } x \neq y\\
& & [P\,/\,x]a(x).Q &= a(x).Q
\end{aligned}$$

Pour transformer une variable globale X en une variable locale x, on calcule cette dernière grâce à une *fonction de hauteur* f qui prend en argument le nom de la variable globale et le terme dans lequel elle est présente : $f : \mathsf{gvar} \to \mathsf{process} \to \mathsf{lvar}$. Étant donnée une telle fonction, la variable locale $f_X(P)$ est la variable qui va servir à abstraire le processus P par rapport à la variable globale X. Par exemple, pour lier X par une réception sur a dans $X \parallel X$, on calcule $x = f_X(X \parallel X)$ et obtient $a(x).(x \parallel x)$.

Suivant ce principe, nous restreignons la syntaxe par un prédicat wf_f paramétré par f définissant l'ensemble des processus dits *bien formés*. Ce sont les processus qui utilisent la fonction de hauteur pour calculer les lieurs et qui n'ont pas de variable locale libre.

- Pour toute variable globale X, nous avons $\mathsf{wf}_f(X)$.
- Si $\mathsf{wf}_f(P)$ alors $\mathsf{wf}_f(\overline{a}\langle P\rangle)$.
- Si $\mathsf{wf}_f(P)$ et $\mathsf{wf}_f(Q)$ alors $\mathsf{wf}_f(P \parallel Q)$.
- Nous avons $\mathsf{wf}_f(\mathbf{0})$.
- Si $\mathsf{wf}_f(P)$ et $x = f_X(P)$, alors $\mathsf{wf}_f(a(x).[x\,/\,X]P)$.

Par la suite, nous notons $\mathsf{abs}\, a\, X\, P$ pour $a(f_X(P)).[f_X(P)\,/\,X]P$. Notons qu'il serait difficile de directement définir les termes bien formés, car ils dépendent de la notion de substitution.

La fonction de hauteur choisie doit bien sûr suivre certains critères afin de ne pas calculer une variable locale déjà utilisée. Nous reprenons les critères de [14] qui garantissent une bonne définition de cette fonction, en y ajoutant un critère supplémentaire (XHC) que nous motivons ci-après.

Pour décrire un de ces critères, nous avons besoin d'une notion supplémentaire : l'ensemble des lieurs ayant dans leur portée une variable globale X donnée, noté $E_X(P)$. Un nom local x est dans $E_X(P)$ si et seulement si P contient un sous terme de la forme $a(x).Q$ avec X présent dans Q. La définition formelle est la suivante.

$$E_X(\mathbf{0}) = \emptyset \qquad E_X(Y) = \emptyset \qquad E_X(P \parallel Q) = E_X(P) \cup E_X(Q)$$

$$E_X(y) = \emptyset \qquad E_X(\overline{a}\langle P\rangle) = E_X(P) \qquad E_X(a(x).P) = \begin{cases} \{x\} \cup E_X(P) & \text{si } X \text{ dans } P\\ \emptyset & \text{sinon} \end{cases}$$

XHE (Équivariance) $f_X(P) = f_{\pi(X)}(\pi(P))$ pour toute permutation π des variables globales. En d'autres termes, la variable calculée de dépend pas des noms choisis pour les variables globales.

XHF (Fraîcheur) $f_X(P) \notin E_X(P)$: la variable calculée n'est pas déjà présente en position liante autour de X dans P.

XHP (Préservation par substitution) si $X \neq Y$ et $X \sharp Q$ alors $f_X(P) = f_X([Q/Y]P)$: la variable calculée ne dépend que des occurrences de X ; changer un sous-terme ne contenant pas X ne change pas la valeur calculée.

XHC (Préservation par congruence) si $P \equiv Q$ alors $f_X(P) = f_X(Q)$: la variable calculée ne dépend pas de l'ordre des compositions parallèles ni de la présence de processus $\mathbf{0}$.

Le critère **XHC** n'est pas présent dans [14] puisque celui-ci porte sur le λ-calcul. Ajouter ce critère permet d'utiliser la congruence structurelle sous les lieurs — c'est une vraie congruence — sans changer leurs valeurs.

Nous supposons désormais que f satisfait **XHE**, **XHF**, **XHP** et **XHC**. Il n'est pas nécessaire de fixer cette fonction, mais nous en donnons un exemple pour montrer son existence. Les variables locales calculées et les variables globales sont des entiers.

$$f_X(X) = 1 \qquad f_X(y) = 0 \qquad f_X(\mathbf{0}) = 0$$
$$f_X(Y) = 0 \quad \text{si } X \neq Y \qquad f_X(\overline{a}\langle P\rangle) = f_X(P)$$
$$f_X(P \parallel Q) = \mathtt{max}(f_X(P), f_X(Q)) \qquad f_X(a(y).P) = \begin{cases} f_X(P) & \text{si } f_X(P) = 0 \text{ ou } f_X(P) > y \\ y+1 & \text{sinon} \end{cases}$$

3.2. Abstractions et agents

Un autre problème n'est pas résolu par la représentation canonique : la règle de la réception. Dans la règle telle qu'elle est présentée ci-dessus, la variable liée est mentionnée dans l'étiquette. Par conséquent, elle doit être prise en compte dans le diagramme de bissimulation « input », où les variables liées par les processus peuvent être différentes, mais sont immédiatement renommées en une nouvelle variable fraîche.

Nous avons préféré suivre une approche à base d'*abstractions*, souvent utilisée pour les calculs d'ordre supérieur. Une abstraction est une réception en attente d'un processus à recevoir.

$$F ::= (x).P \;\Big|\; F \parallel P \;\Big|\; P \parallel F$$

Étant donnée une abstraction, on peut l'instancier en parcourant l'abstraction pour retrouver le cas de base $(x).P$ qui indique la variable x à substituer dans le processus P. Notre définition, que nous appelons *abstraction localisée*, est différente de l'approche classique où une abstraction a exactement la forme $(x)P$. Dans ce cas, il est nécessaire de définir comment faire remonter le lieur au dessus des autres constructeurs pour retrouver la syntaxe attendue. Nous avons trouvé que les abstractions localisées étaient plus pratiques à manipuler. En effet, déplacer un lieur implique que l'on recalcule la valeur de la variable liée, ce qui n'a aucune utilité puisque celle-ci est immédiatement instanciée. L'utilisation d'abstraction localisée est très similaire à l'approche proposée dans [17].

Formellement, l'instanciation d'une abstraction, notée $F \bullet P$, est définie par induction sur la structure de l'abstraction.

$$(x).P \bullet P' = [P' \,/\, x]P \qquad (F \parallel P) \bullet P' = (F \bullet P') \parallel P \qquad (P \parallel F) \bullet P' = P \parallel (F \bullet P')$$

On définit enfin un *agent*, qui est soit un processus soit une abstraction : $A ::= P \;\Big|\; F$.

On adapte en conséquence la sémantique de HOCore. Le système de transition étiqueté transforme ainsi des processus en agents grâce aux règles suivantes :

Inp $a(x).P \xrightarrow{a} (x).P$;
Out $\overline{a}\langle P\rangle \xrightarrow{\overline{a}\langle P\rangle} \mathbf{0}$;
Act1 si $P \xrightarrow{\alpha} A$ alors $P \parallel Q \xrightarrow{\alpha} A \parallel Q$;
Tau1 si $P \xrightarrow{\overline{a}\langle P''\rangle} P'$ et $Q \xrightarrow{a} F$ alors $P \parallel Q \xrightarrow{\tau} P' \parallel (F \bullet P'')$;
Act2 et Tau2 sont les règles symétriques.

On remarque que les abstractions suppriment tout besoin de condition de garde pour la règle Act1.

3.3. Enrichissement du LTS

Un dernier changement a permis de simplifier la notion de bissimulation : nous avons intégré la construction $\mathsf{remove}_X(P, P')$ dans le LTS en ajoutant la règle :
REM $X \xrightarrow{X} \mathbf{0}$.

Nous avons ainsi $\mathsf{remove}_X(P, P')$ si et seulement si $P \xrightarrow{X} P'$. Nos définitions des relations « input », « output » et « variable » deviennent les suivantes.

- Une relation $\mathcal{R}$ est dite *input* si dès que $P \mathcal{R} Q$ et $P \xrightarrow{a} F$, alors il existe F' tel que $Q \xrightarrow{a} F'$ et pour tout X frais pour P et Q nous avons $F \bullet X \mathcal{R} F' \bullet X$.
- Une relation $\mathcal{R}$ est dite *output* si dès que $P \mathcal{R} Q$ et $P \xrightarrow{\overline{a}\langle P''\rangle} P'$, alors il existe Q' et Q'' tels que $Q \xrightarrow{\overline{a}\langle Q''\rangle} Q'$, $P' \mathcal{R} Q'$ et $P'' \mathcal{R} Q''$.
- Une relation $\mathcal{R}$ est dite *variable* si $P \mathcal{R} Q$ et $P \xrightarrow{X} P'$ implique qu'il existe Q' tel que $Q \xrightarrow{X} Q'$ et $P' \mathcal{R} Q'$.

Notons que même si l'approche canonique nous permet de définir proprement la gestion de l'α-conversion, nous n'échappons pas aux difficultés standards du π-calcul (relations « early » ou « late ») qui requièrent une variable fraîche dans le cas « input ». Ceci n'est pas gênant en pratique : nous montrons que la quantification universelle sur la variable fraîche peut être remplacée par une quantification existentielle (section 4.8), ce qui est nécessaire pour prouver la décidabilité de l'IO-bissimilarité (section 4.12).

4. Développement en Coq

Nous détaillons maintenant le développement en Coq, indiquant quels résultats et définitions sont présents dans quels fichiers. [3] Les auteurs étant novices en Coq, les preuves sont longues et peu structurées et pourraient être grandement améliorées.

4.1. HOC01Defs.v

Nous commençons par définir la syntaxe des processus, sachant que lvar, gvar et chan sont des entiers naturels.

```
Inductive process : Set :=
| Send : chan →process →process
| Receive : chan →lvar →process →process
| Lvar : lvar →process
| Gvar : gvar →process
| Par : process →process →process
```

3. Le développement est disponible en ligne : `http://sardes.inrialpes.fr/~aschmitt/research/hocore/toc.html`.

Nous définissons ensuite les fonctions de bases (GV calcule l'ensemble des variables globales d'un processus, size sa taille, ...) et la congruence structurelle. La fonction subst $P\ X\ Q$ calcule $[P\ /\ X]Q$ sans se soucier de capture potentielle.

```
Fixpoint subst (p:process) (X:gvar) (q:process) : process :=
  match q with
    | Send a q ⇒Send a (subst p X q)
    | Receive a x q ⇒Receive a x (subst p X q)
    | Lvar _ ⇒q
    | Gvar Y ⇒if beq_nat X Y then p else q
    | Par q1 q2 ⇒(Par (subst p X q1) (subst p X q2))
    | Nil ⇒Nil
  end.
```

Enfin, les dernières définitions sont celles de la représentation canonique : height_fun est le type des fonctions de hauteur ; XHE, XHF, XHP, XHC et good sont des prédicats sur ces fonctions (good étant la conjonction des quatre prédicats). La fonction abs permet de facilement construire des abstractions.

```
Definition abs (f:height_fun) (a:chan) (X:gvar) (p:process) :=
  Receive a (f X p) ([Lvar (f X p)//X]p).
```

Par exemple, le processus `abs a X (Gvar X)` correspond, en utilisant la fonction de hauteur de la section 3.1, au processus $a(1).1$. On peut ainsi écrire directement des processus dans une syntaxe proche de la syntaxe informelle.

Nous pouvons maintenant définir ce qu'est un processus bien formé.

```
Inductive wf (f:height_fun) : process → Prop :=
| WfSend : ∀(a:chan) (p:process), (wf f p) → (wf f (Send a p))
| WfReceive : ∀(a:chan) (x:lvar) (X:gvar) (p:process), (wf f p) → x=(f X p) →
              (wf f (Receive a x ([Lvar x//X]p)))
| WfGvar : ∀(X:gvar), (wf f (Gvar X))
| WfPar : ∀(p q:process), (wf f p) → (wf f q) → (wf f (Par p q))
| WfNil : (wf f Nil).
```

4.2. HOC02DefLTS.v

Nous définissons dans ce fichier le système de transitions étiquetées, avec les concepts afférents comme les abstractions, les agents ou l'instanciation.

4.3. HOC03FreshLemmas.v et HOC03SizeLemmas.v

Nous prouvons dans ces fichiers des petits lemmes portant sur la fraîcheur des variables ou sur la taille des processus. Par exemple, on y montre que l'on ne manquera pas de variables fraîches.

```
Lemma find_fresh_gvar1 : ∀p, ∃X, X#p.
```

Les lemmes sur la taille portent à la fois sur la taille des processus, mais également sur le nombre d'occurrences de variables. Les constructeurs AP et AA plongent respectivement les processus et les abstractions dans le type des agents.

```
Lemma size_remove : ∀X p p', p—X→(AP p') → size p = S (size p').
```

```
Lemma sizeX_GV : ∀X p, In X (GV p) ↔ sizeX X p > 0.
```

4.4. HOC04SubstLemmas.

Un aspect crucial de la construction canonique est la définition des différentes substitutions. La première remplace une variable globale par un processus, et est notée en Coq $[q//X]p$. La deuxième remplace une variable locale par un processus, et est notée $[q///x]p$. La dernière permute deux variables globales dans et un processus et est notée $\{X, Y\}P$. Nous donnons, entre autres, des lemmes qui décrivent comment les substitutions commutent. Les lemmes les plus importants de cette section sont directement dérivés de [14], comme le lemme suivant.

```
Lemma substs_commute : ∀(x:lvar) (X Y:gvar) (p q:process),
  X≠Y → x#q → [(Gvar Y)///x] ([q//X] p) = [q//X] ([(Gvar Y)///x] p).
```

4.5. HOC05CongrLemmas.v

Nous prouvons ici des lemmes sur la congruence structurelle, comme les deux lemmes suivants. Bien que nous utilisions dans cet article la notation ≡ pour la congruence structurelle, dans le développement en ligne, ≡ est une étape de congruence structurelle et sa clôture réflexive transitive est notée $\equiv*$.

```
Lemma congr_subst : ∀(p p':process), p≡p' → ∀X q q', q≡q' → [q//X]p ≡ [q'//X]p'.

Lemma congr_fresh : ∀p p' X, p≡p'→  X#p → X#p'.
```

4.6. HOC06CanonicalLemmas.v

Nous prouvons des lemmes sur la représentation canonique. Le lemme suivant spécifie que l'on peut remplacer dans le corps d'une réception la variable locale liée par une variable globale fraîche et obtenir un processus qui reste bien formé. Ce lemme sert pour la propriété « input » des IO-bissimulations.

```
Lemma wf_receive : ∀a x p, wf f (Receive a x p) →
  ∀X, X#p → wf f ([Gvar X///x]p) ∧ Receive a x p = abs f a X ([Gvar X///x]p).
```

Nous prouvons également l'existence d'une fonction de hauteur, en montrant que la fonction suivante satisfait les quatre critères XHE, XHF, XHP et XHC.

```
Fixpoint GoodF X P :=
  match P with
    | Gvar Y ⇒if beq_nat X Y then 1 else 0
    | Lvar _ ⇒0
    | Nil ⇒0
    | Par P1 P2 ⇒max (GoodF X P1) (GoodF X P2)
    | Send a P ⇒GoodF X P
    | Receive a x P ⇒let m' := GoodF X P in
      if beq_nat m' 0 then 0 else if lt_dec x m' then m' else S x
  end.
```

4.7. HOC07TransLemmas.v

Lors de preuves de bissimulations, il est très utile de pouvoir dériver certaines informations de l'existence de transition. Par exemple, si un processus a une réduction interne, il possède également deux transitions successives, une émission et une réception, qui mènent au même résultat.

```
Lemma decomposition_tau1 : ∀p p', p—τ→(AP p') →
  ∃a, ∃p1, ∃p'', ∃fp, p—ā⟨p''⟩→(AP p1) ∧ p1—a→(AA fp) ∧ p' = inst_abs fp p''.
```

Le lemme inverse est encore plus important : une émission suivie d'une réception sur le même nom indique qu'une réduction interne est possible. Il est crucial pour que ce lemme soit vrai que le calcul soit asynchrone — qu'il n'y ait pas de continuation à l'émission d'un message — afin de montrer que la réception est active dès le processus initial.

```
Lemma decomposition_tau2 : ∀a p p1 p'' fp,
  p—ā⟨p''⟩→(AP p1) ∧ p1—a→(AA fp) → p—τ→(AP (inst_abs fp p'')).
```

4.8. HOC08Bisimulation.v

Nous définissons la notion de IO-bissimilarité en tant que plus grande relation satisfaisant les critères d'IO-bissimulation.

```
Definition in_relation (R:relation process) : Prop := ∀p q, (R p q) →∀a fp,
  p—a→(AA fp) →∃fq, (q—a→(AA fq) ∧
                        (∀ X, X#p →X#q →(R (inst_abs fp (Gvar X)) (inst_abs fq (Gvar X))))).

Definition out_relation (R:relation process) : Prop :=
  ∀p q, (R p q) →∀a p' p'', p—ā⟨p''⟩→(AP p') →∃q', ∃q'', (q—ā⟨q''⟩→(AP q') ∧ (R p' q') ∧ (R p'' q'')).

Definition var_relation (R:relation process) : Prop :=
  ∀p q, (R p q) →∀X p', p—X→(AP p') →∃q', q—X→(AP q') ∧ (R p' q').

Definition IO_bisimulation (R:relation process) : Prop :=
  (Symmetric R) ∧ (in_relation R) ∧ (out_relation R) ∧ (var_relation R).

Definition IObis p q : Prop := ∃R, (IO_bisimulation R) ∧ (R p q).
```

Nous montrons que cette définition est cohérente en prouvant que l'IO-bissimilarité est une IO-bissimulation. Nous prouvons également que l'IO-bissimilarité est réflexive, symétrique et transitive. Ce dernier point n'est vraiment pas trivial à cause de la quantification universelle de la variable X dans le cas de l'input : on pourrait avoir une variable fraîche par rapport à P et R qui n'est pas fraîche par rapport à Q, avec $P \sim Q \sim R$. Nous montrons que cela ne peut pas être le cas grâce au lemme suivant, qui indique que des processus IO-bissimilaires ont les mêmes variables fraîches.

```
Lemma gfresh_IObis : ∀X p q, p∼q →X#p →X#q.
```

Nous étudions ensuite une définition alternative pour la relation « input », où le quantificateur universel pour la variable fraîche devient existentiel, et nous montrons que l'IO-bissimilarité « existentielle » coïncide avec l'IO-bissimilarité.

```
Definition in_relation_ex (R:relation process) : Prop := ∀p q, (R p q) →∀a fp,
  p—a→(AA fp) →∃fq, (q—a→(AA fq) ∧
                        (∃ X, X#p ∧ X#q ∧ (R (inst_abs fp (Gvar X)) (inst_abs fq (Gvar X))))).
```

Nous montrons enfin que la congruence structurelle est incluse dans l'IO-bissimilarité.

4.9. HOC09Guarded.v

L'IO-bissimilarité restreinte aux processus bien formés est préservée par substitution.

```
Lemma IObis_subst : ∀X p q r r',
  (wf f p) → (wf f q) → (wf f r) → (wf f r') → p∼q → r∼r' → ([r//X]p) ∼ ([r'//X]q).
```

La preuve de ce lemme n'est pas immédiate et demande l'introduction du concept de variable gardée. La variable X est gardée dans P si X n'apparaît qu'à l'intérieur des émissions et des réceptions — X n'apparaît pas « à toplevel ». On montre ensuite que l'on peut décomposer tout processus en un processus où la variable X est gardée mis en parallèle avec une composition parallèle de X.

4.10. HOC10TauBis.v

Nous prouvons ici que l'IO-bissimilarité est une τ-bissimulation.

```
Theorem IObis_TauBis : tau_bisimulation IObis.
```

4.11. HOC11Barbed.v

Nous montrons le premier théorème qui nous intéresse : l'IO-bissimilarité restreinte aux processus bien formés implique la congruence barbue. Pour ce faire, nous définissons la notion de contextes bien formés et montrons que l'IO-bissimilarité est close sous ces contextes.

```
Definition context_closed (R:relation process) : Prop :=
  ∀p q, wf f p → wf f q → (R p q) → ∀C, (wf_ctxt f C) → (R (fill C p) (fill C q)).

Definition equiv p q : Prop := ∃R, (Symmetric R) ∧ (tau_bisimulation f R) ∧
  (context_closed R) ∧ (out_barb_preserving R) ∧ (R p q).

Theorem IObis_correct : ∀p q, wf f p → wf f q → IObis p q → equiv p q.
```

4.12. HOC12Decidability.v

Nous montrons enfin le deuxième théorème qui nous intéresse : l'IO-bissimilarité est décidable. Pour ce faire, nous implémentons la procédure décrite en section 2.5, notée `bio`, et montrons que cette procédure est adaptée.

```
Lemma bio_ok : ∀p q, wf f p → wf f q → (p ∼ q ↔ bio p q = true).

Theorem IObis_decidable : ∀p q, wf f p → wf f q → decidable (p ∼ q).
```

5. Travaux connexes

Parmi les formalisations de calculs de processus, on peut particulièrement remarquer les travaux de Parrow et al sur le psi-calcul implémentés en Isabelle [4]. Même si l'ordre supérieur ne pose pas de difficulté en soit, il n'est pas clair que leur approche peut être directement adaptée sans passer par un encodage dans un calcul sans ordre supérieur. Le π-calcul a été également formalisé en Coq, en utilisant des indices de De Bruijn [5] ou une syntaxe abstraite d'ordre supérieur [6].

Les langages d'ordre supérieur formalisés sont souvent des variantes du λ-calcul comme le système F [3]. De nombreuses formalisations ont ainsi été proposées dans le cadre du défi PoplMark [1]. La principale différence avec notre travail porte sur les propriétés qui sont démontrées.

Nous nous sommes positionnés par rapport aux approches nominales et à base d'indices de De Bruijn en 3.1. D'autres travaux, plus récents, proposent un modèle encore plus fondamental des lieurs. Nous aurions pu par exemple utiliser le cadre générique des travaux de Pouillard et Pottier [15], mais non seulement il n'est pas clair qu'un tel niveau d'abstraction nous est utile, de plus il manque encore à cette approche des techniques pour raisonner sur les termes.

6. Conclusion et travaux futurs

Nous avons présenté une formalisation du calcul de processus HOCore et la preuve de certaines de ses propriétés dans l'assistant de preuve Coq. Pour obtenir une définition formelle du calcul, nous avons précisé sa sémantique en utilisant plusieurs techniques telles que la représentation canonique locale des lieurs et l'utilisation d'abstractions localisées. Nous avons montré que l'IO-bissimilarité est incluse dans la congruence barbue et qu'elle est décidable. Ces travaux ont permis de rendre plus précis les résultats de [8] tout en essayant de donner une intuition sur la cause de la décidabilité de la congruence barbue. Malgré notre peu d'expérience en Coq, nous avons été agréablement surpris de ne pas rencontrer de difficulté technique majeure, nous laissant supposer que cette approche est adaptée.

Nous continuons à travailler sur ce développement, en suivant plusieurs directions. Nous souhaitons montrer que toutes les notions usuelles de bissimulations coïncident, en particulier que l'IO-bissimilarité est complète par rapport à la congruence barbue. La preuve de ce résultat sera sûrement complexe, car elle utilise de nombreux raisonnements peu formels, de la forme « nous renommons toutes les émissions de messages en des émissions de messages sur des noms frais ». Il sera également intéressant de montrer l'axiomatisation de la congruence barbue qui fait appel à des notions de décomposition en produits de facteurs premiers des processus.

Nous voulons bien entendu dépasser le minimalisme de HOCore et nous assurer que notre approche continue à fonctionner lorsque l'on ajoute des primitives au calcul. La première extension consiste à ajouter la restriction de nom $\nu a.P$ à HOCore, ce qui conduit au π-calcul d'ordre supérieur. Comme l'opérateur de réception, la restriction est un lieur et les questions d'α-équivalence doivent être traitées. L'approche classique de la restriction de nom est de la rendre extrêmement flexible et d'ajouter des règles dans l'équivalence structurelle lui permettant de commuter avec l'opérateur parallèle et elle-même : $P \parallel \nu a.Q \equiv \nu a.P \parallel Q$ si a n'est pas libre dans P, $\nu a.\nu b.P \equiv \nu b.\nu a.P$. Si nous prenons une approche localement canonique avec ces règles, une conséquence sera que les valeurs des lieurs pourront changer lors de leur application. C'est pourquoi nous envisageons une sémantique différente, où l'opérateur de restriction de nom est en fait un opérateur de création de nom frais quand il arrive en contexte d'exécution. Ces sémantiques sont équivalentes — en l'absence de passivation, voir ci-dessous —, car cela revient à faire monter tous les opérateurs de restriction en contexte d'évaluation à la racine du terme. Nous pourrons ensuite utiliser une approche à base de bissimulations environnementales pour mettre les processus en relation [12].

Une deuxième extension que nous souhaitons considérer est l'ajout de l'opérateur de passivation [9]. Cet opérateur, basé sur une notion de *localité*, permet d'interrompre et de capturer un processus en cours d'exécution. Nous conjecturons qu'ajouter seulement cet opérateur ne casse pas la décidabilité de la congruence barbue, mais ceci n'a pas encore été démontré. De plus, certaines preuves portant sur des calculs avec passivation et restriction sont très complexes [10]. Il serait très utile de pouvoir valider ces preuves en Coq et nous assurer que notre approche passe à l'échelle.

Remerciements

Nous tenons à remercier Damien Pous et Thomas Braibant pour leur aide et leur disponibilité lors de notre apprentissage de Coq, ainsi que les rapporteurs anonymes pour leurs remarques éclairées.

Références

[1] B. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich et S. Zdancewic : Mechanized metatheory for the masses : The PoplMark challenge. *In TPHOLs*, p. 50–65, 2005.

[2] B. Aydemir, A. Bohannon et S. Weirich : Nominal reasoning techniques in coq. *In International Workshop on Logical Frameworks and Meta-Languages :Theory and Practice (LFMTP)*, Seattle, WA, USA, août 2006.

[3] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack et S. Weirich : Engineering formal metatheory. *In ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, p. 3–15. ACM, jan. 2008.

[4] J. Bengtson et J. Parrow : Psi-calculi in isabelle. *In Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, p. 99–114, Berlin, Heidelberg, août 2009. Springer-Verlag.

[5] D. Hirschkoff : A full formalisation of pi-calculus theory in the calculus of constructions. *In Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, vol. 1275, p. 153–169, Murray Hill, NJ, USA, août 1997. Springer.

[6] F. Honsell, M. Miculan et I. Scagnetto : pi-calculus in (co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, fév. 2000.

[7] B. Huffman et C. Urban : Proof pearl : A new foundation for nominal isabelle. *In Proceecings of the 1st Conference on Interactive Theorem Proving*, vol. 6172 de *LNCS*, p. 35–50, Edinburgh, UK, juil. 2010. Springer Verlag.

[8] I. Lanese, J. A. Pérez, D. Sangiorgi et A. Schmitt : On the expressiveness and decidability of higher-order process calculi. *Information and Computation*, 209(2):198–226, fév. 2011.

[9] S. Lenglet : *Bisimulations dans les calculs avec passivation.* Thèse de doctorat, Université de Grenoble, 2010.

[10] S. Lenglet, A. Schmitt et J.-B. Stefani : Characterizing contextual equivalence in calculi with passivation. *Information and Computation*, 209(11):1390–1433, nov. 2011.

[11] The Coq development team : *The Coq proof assistant reference manual*, 2009. Version 8.3.

[12] A. Piérard et E. Sumii : Sound bisimulations for higher-order distributed process calculus. *In Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011*, vol. 6604 de *LNCS*, p. 123–137. Springer, 2011.

[13] A. M. Pitts : Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[14] R. Pollack, M. Sato et W. Ricciotti : A canonical locally named representation of binding. *Journal of Automated Reasoning*, p. 1–23, mai 2011. 10.1007/s10817-011-9229-y.

[15] N. Pouillard et F. Pottier : A fresh look at programming with names and binders. *In Proceedings of the fifteenth ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, p. 217–228, sept. 2010.

[16] D. Sangiorgi : Bisimulation for higher-order process calculi. *Information and Computation*, 131(2):141–178, dec 1996.

[17] A. Tiu et D. Miller : Proof search specifications of bisimulation and modal logics for the pi-calculus. *ACM Transactions on Computational Logic (TOCL)*, 11:13 :1–13 :35, January 2010.