A Constructive Proof of the Soundness of the Encoding of Random Access Machines in a Linda Calculus with Ordered Semantics.

Claudio Sacerdoti Coen*

Department of Computer Science
Via di Mura Anteo Zamboni 7, 40127 Bologna, ITALY.
sacerdot@cs.unibo.it

Abstract. Random Access Machines (RAMs) are a deterministic Turing-complete formalism especially well suited for being encoded in other formalisms. This is due to the fact that RAMs can be defined starting from very primitive concepts and operations, which are unbounded natural numbers, tuples, successor, predecessor and test for equality to zero. Since these concepts are easily available also in theorem-provers and proof-assistants, RAMs are good candidates for proving Turing-completeness of formalisms using a proof-assistant. In this paper we describe an encoding in Coq of RAMs into a Linda Calculus endowed with the Ordered Semantics. We discuss the main difficulties that must be faced and the techniques we adopted to solve them.

1 Introduction

A Random Access Machine (RAM) [12] is a computational model composed of a finite set of registers and of a program, that is a sequence of simple numbered instructions, like arithmetical operations (on the content of registers) or conditional jumps. A register holds an unbounded natural number. The state of a RAM is the set of values stored in each register plus the program counter (PC), which is the address of the next instruction to execute. The computation starts from an initial state and proceeds iterating the two operations of fetching and executing the next instruction (determined by the PC). The computation ends as soon as the value of the PC is greater than the index of the last instruction.

The RAMs formalism is Turing complete and has several characteristics that make it particularly appealing for being encoded into other formalisms to show their Turing completeness. First of all a RAM can be defined using only very primitive concepts and operations, which are unbounded natural numbers, tuples, successor, predecessor and test for equality to zero. These concepts are likely to be already provided in any formalism which we want to show Turing complete. Moreover it is possible to reduce the set of instructions to only two elements, `Succ` (which increments a register) and `DecJump` which decrements a

* Work partially supported by the European Project IST-33562 MoWGLI

register if it is not zero or jumps to a given address otherwise. As a consequence the proofs of completeness of the encodings are very small, having to address only a couple of cases and only three possible transitions. Finally, RAMs are completely deterministic, which prevents the usual difficulties given by the simulation of non-deterministic systems into deterministic ones.

Just to mention a couple of examples, RAMs have been successfully encoded into several process algebras, such as asynchronous CCS [13] and the ambient calculus [14]. Even if providing the encoding can be challenging, the soundness and completeness proofs are usually given in just a few lines, and heavily rely on the reader intuition to figure out the details by case reasoning.

In the last few years, several process calculi have been formalized, using proof-assistants, in both constructive and classical logics. Surprisingly, to our knowledge nobody tried to reproduce a proof of Turing-completeness of those calculi. Nevertheless, when we started this work, we used to think that giving one of those proofs by means of a sound and complete encoding of RAMs into the calculus would have been completely trivial. The reason of our belief was that the same characteristics that make RAMs so well suited for their encoding in other formalisms also make their definition in a logical framework quite straightforward. Moreover the soundness proof would just be a laborious induction on the derivation.

In this paper we present our experience in using the proof-assistant Coq [2] to provide an encoding of RAM into a Turing complete process algebra. We also give a constructive proof of the soundness property of the encoding. As usual, the soundness proof exploits the completeness property. However, we axiomatically assume the completeness of the calculus, that can be proved by induction over the structure of RAMs and that of their derivations. Since the RAM formalism is deterministic, the required induction is almost straightforward and presents no major challenge, even if it is extremely laborious and time consuming. The soundness proof quickly turned out to require a major effort and its proof-script is currently more than 10.000 lines long and it is still under development, since we plan to provide soon also the proof of completeness. The up-to-date set of Coq files are available for download at the following address: `http://www.cs.unibo.it/RAMs_to_Linda.tgz`

In section 2 we briefly introduce the Coq proof-assistant and the main features of its expressive logic. In section 3 we review the usual definition of RAMs and we present its formal definition in the Coq system. In section 4 we introduce a minimal process algebra built around the Linda primitives [5], originally proposed in [1]. The formal definition in Coq is also provided. In section 5 we present the encoding of RAMs into the Linda Calculus of section 4 and we outline the formal proof of soundness. Finally, in section 6 we present our conclusions and future work proposal.

2 The Coq Proof Assistant

Coq [2] is one of the most advanced proof-assistants, used in Europe both in academical and industrial settings. It is based on an expressive logical framework, the Calculus of (Co)Inductive Constructions (CIC), which allows both constructive and classical proofs and that is particularly suitable for proving the correctness of programs with respect to their specifications, with the additional possibility of automatically extracting a certified program from a constructive proof of the existence of a function that satisfies a given specification.

Briefly, a Coq development is made of a list of definitions, inductive definitions and theorems whose correctness is checked by the system. Inductive definitions, which are one of the main features of CIC, can be used to define both inductive data types (lists, trees, natural numbers) and inductive properties (by giving the set of inference rules that define the property). It is possible to define by structural recursion total functions over an inductive data-type (in the style of the system T); in the same way it is also possible to prove by structural induction properties over an inductive data-type. A primitive notion of case analysis over inductive defined terms is also provided.

Since the main topic of the paper is the technique used for the encoding of the soundness proof in Coq, the interested reader needs a general expertise in the world of proof-assistants and in the Coq system in particular. Due to lack of space, the required knowledge can not be provided in this paper.

3 Random Access Machines

The following is the usual presentation of a particular class of RAMs with only two-instructions. The proof of their Turing completeness was given in [11].

Definition 1. *A* Random Access Machine *(RAM) is a pair* $(\boldsymbol{r}, \boldsymbol{p})$ *where* $\boldsymbol{r} = (r_1, \ldots, r_n)$ *is the set of registers and the program* $\boldsymbol{p} = (i_1, \ldots, i_m)$ *is a finite list of instructions. Each instruction can have one of the following two forms:*

- $\mathtt{Succ}(r_j)$*: adds 1 to the content of register* r_j
- $\mathtt{DecJump}(r_j, s)$*: if the content of register* r_j *is not zero, then decreases it by 1 and go to the next instruction; otherwise jumps to instruction* s

Definition 2. *Let* R *be a RAM whose registers are* $(r_1, \ldots, r_n)$*. A* configuration *of* R *is a tuple* $(p, c_1, \ldots, c_n)$ *where the program counter (PC)* p *is the index of the next instruction to be executed and* c_l *is the content of the register* r_l*.*

Definition 3. *Let* R *be a RAM whose registers are* $(r_1, \ldots, r_n)$ *and whose program is* $(i_1, \ldots, i_m)$*. A configuration* $(p, c_1, \ldots, c_n)$ moves *into a configuration* $(q, c'_1, \ldots, c'_n)$ *(notation:* $(p, c_1, \ldots, c_n) \longrightarrow_R (q, c'_1, \ldots, c'_n)$*) iff:*

- $i_p = \mathtt{Succ}(r_j)$ *and* $c'_j = c_j + 1$ *and* $\forall l \neq j.\ c'_n = c_n$ *and* $q = p + 1$
- $i_p = \mathtt{DecJump}(r_j, s)$ *and* $c_j > 0$ *and* $c'_j = c_j - 1$ *and* $\forall l \neq j.\ c'_n = c_n$ *and* $q = p + 1$

– i_p = *DecJump*(r_j, s) *and* $c_j = 0$ *and* $\forall l.\, c'_n = c_n$ *and* $q = s$

Definition 4. *A configuration* $(p, c_1, \ldots, c_n)$ *is* stuck *or* final *when it cannot move. This happens only when* $p > m$*, where* m *is the length of the program.*

As usual, given a RAM R and an initial configuration C, we say that C reduces to C' when $C \longrightarrow_R^* C'$ and C' is final, where $\longrightarrow_R^*$ is the reflexive and transitive closure of $\longrightarrow_R$.

3.1 RAMs in Coq

The only non-trivial issue on the formalization of RAMs in Coq is the way registers are encoded. In fact the tuple $(r_1, \ldots, r_n)$ of registers just plays the role of a finite set of indexes such that no instruction refers to a non-existent index. The only additional constraint we need to impose on the set is that there exists a decidable equality relation over its elements.

Since the list of instructions is finite and since registers not referred by any instruction play no role during the reduction, we could have used just the type of natural numbers for register indexes (with no concern about the fact that the set of indexes is actually infinite). In this case it would have been possible in Coq to define a function to compute the greatest used register index. Instead we preferred to stick ourselves to this alternative definition:

```
Definition register := nat.

Inductive instruction : Set :=
   Succ : register→instruction
 | DecJump : register→nat→instruction.

Inductive uses_at_most [r:register] : instruction→Prop :=
   okSucc : ∀r':register.(r'<r)→(uses_at_most r (Succ r'))
 | okDecJump :
     ∀r':register.∀n:nat.(r'<r)→(uses_at_most r (DecJump r' n)).

Record RAM : Set :=
  { registers_number : nat ;
    program : (list instruction) ;
    program_ok : (AllS (uses_at_most registers_number) program)
  }.
```

i.e. a RAM is a record where the number of available registers is made explicit and a proof is provided of the fact that each instruction in the program does not refer to any unavailable register. We prefer this formalization for several reasons:

– This formalization is closer to the traditional one and does not identify two RAMs with the same program, but a different set of available registers. Unused but still available registers play a role in the traditional proofs of some operations used to combine or derive RAMs from other RAMs.

– The fact that the number of available registers is invariant during the reduction holds trivially for RAMs, but can be much more difficult to prove when RAMs are encoded into other calculi. That invariant plays an important role in the soundness and completeness proofs. For example, in section 5 we will see that we need to encode in the same data-type (i.e. natural numbers) both the indexes of registers and the value of the program counter (which is an unbounded natural number). The easiest way to do that is to encode each program counter value p as $M + p$ where M is the number of available registers. The decoding function relies on the fact that M does not change during the reduction.

We must now formalize configurations. In particular we must choose a representation of the state of the registers. The choice has a great impact on the way two configurations are checked for equality: we may choose an encoding where two configurations are equal iff they are equal using Leibniz equality; or we can provide an ad-hoc equality predicate. The first scenario seems at first more appealing, since Leibniz equality is better handled by Coq's tactics with respect to a user-provided one. Moreover, a possible implementation could be using ordered lists of natural numbers of length n. Unfortunately, reasoning on ordered lists of fixed length may complicate some proofs and introduce the usual problems related to the fact that every operation working on them (and, in particular, the state update function) must be proof-irrelevant on the proof argument. Proof-irrelevance and dependent types do not combine easily in intensional proof theories as the one of Coq [3, 4].

A good compromise is using functions from natural numbers to natural numbers, using extensional equality to compare the content *of available registers only*:

```
Record configuration : Set :=
 { program_counter : nat ;
   registers : register→nat
 }.

Definition RAM_configuration_equality :
 nat→configuration→configuration→Prop
:=
 λregisters_no:nat.λH,H':configuration.
  Cases H of
   (Build_configuration program_counter registers)⇒
    Cases H' of
     (Build_configuration program_counter' registers')⇒
      program_counter=program_counter'∧
       ∀r:register.
        (r<registers_no)→
         (registers r)=(registers' r)
    end
  end.
```

The relation $\longrightarrow_R$ is given as an inductive predicate `step`. Its definition and some of its properties (determinism, decidability of being stuck, non modification of unavailable registers) as well as the definition of $\longrightarrow_R^3$ and its basic properties are straightforward.

Since we have just defined an encoding of RAMs into CIC, we should prove the adequacy of the encoding; a detailed proof would be quite complex since it heavily relies on the metatheory of CIC. Nevertheless, the usual sources of complexity that are involved in the encodings of calculi — as the handling of bound variables and α-conversion — are not raised by RAMs, due to the simplicity of the formalism. As a consequence, the adequacy property is not much interesting and it is essentially pretty obviously granted by construction. The same remark holds also for the Linda calculus that we are going to present in the next section.

The whole definition of RAMs, their operational semantics and the proof of these basic properties are provided in the file `RAMs.v` which is just 366 lines long.

4 The Linda Calculus

Linda [5] is the most prominent representative of the family of coordination languages based on the principle of *generative communication* [5, 6]: a sender communicates with a receiver through a shared data space (called *tuple space* TS) where emitted messages are collected; the receiver can read the message or even remove it from the TS; a message generated by a process has an independent existence in the TS and can be concurrently manipulated by any process.

The Linda coordination primitives are meant to be embedded into an already existent programming language. Moreover their semantics is not specified formally and there are several reasonable interpretations for the semantics of some operations. In particular, it is unclear if a sender who is emitting a message should block until the message actually appears in the tuple space (*ordered semantics*) or is free to continue immediately its execution (*unordered semantics*).

Busi, Gorrieri and Zavattaro defined in [1] a minimal process algebra containing the coordination primitives of Linda together with parallel composition and a limited form of recursion (the replication operator [10] guarded on inputs). The algebra is named *Linda Calculus*. In the same paper they proved that the Linda Calculus is Turing complete only when the ordered semantics is assumed. The first part of the proof is made showing a sound and complete embedding of RAMs in the Linda Calculus with ordered semantics. The proof of soundness and completeness is just three lines long in the full version of the paper.

In this section we present a formalization in Coq of the Linda Calculus presented in [1]. The next section is devoted to the formalization of the encoding.

Let $Messages$ be a denumerable set of message names, ranged over by $a, b, \ldots$ The syntax of the Linda Calculus is defined by the following grammar:

$$P \ ::= \ < a > \ \mid C \mid P|P$$

$$C \ ::= \ 0 \mid \eta.C \mid \mu?C\_C \mid C||C$$

(1)	$< a > \xrightarrow{\overline{a}} 0$	(2)	$in(a).P \xrightarrow{a} P$
(3)	$rd(a).P \xrightarrow{\underline{a}} P$	(4)	$!in(a).P \xrightarrow{a} P \mid !in(a).P$
(5)	$inp(a)?P\_Q \xrightarrow{a} P$	(6)	$rdp(a)?P\_Q \xrightarrow{\underline{a}} P$
(7)	$inp(a)?P\_Q \xrightarrow{\neg a} Q$	(8)	$rdp(a)?P\_Q \xrightarrow{\neg a} Q$
(9)	$P \| Q \xrightarrow{\tau} P \mid Q$	(10)	$out(a).P \xrightarrow{\tau} < a > \mid P$
(11)	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \alpha \neq \neg a$	(12)	$\dfrac{P \xrightarrow{\neg a} P' \quad Q \stackrel{\overline{a}}{\not\longrightarrow}}{P \mid Q \xrightarrow{\neg a} P' \mid Q}$
(13)	$\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$	(14)	$\dfrac{P \xrightarrow{\underline{a}} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q}$

Table 1. Operational semantics (symmetric rules omitted).

where:

$$\eta \ ::= \ in(a) \mid rd(a) \mid out(a) \mid !in(a)$$

$$\mu \ ::= \ inp(a) \mid rdp(a)$$

Agents, ranged over by $P, Q, \ldots$, consist of the parallel composition of the messages already in the TS (each one denoted by one agent $< a >$) and the concurrent programs denoted by $C, D, \ldots$, that share the tuples. A program can be a terminated program 0, a prefix form $\eta.P$, an *if-then-else* form $\mu?P\_Q$, or the parallel composition of programs. A prefix η can be one of the Linda primitives $in(a)$, $rd(a)$ or $out(a)$ indicating the withdrawing, the reading (without consumption) or the emission of message a, respectively. We also consider the bang operator $!in(a)$ which is a form of replication guarded on input operations. The if-then-else forms are used to model the $inp(a)$ and $rdp(a)$ Linda primitives: $inp(a)?P\_Q$ ($rdp(a)?P\_Q$) is a program which requires the message a to be consumed (or simply read); if a is present, the program P is executed, otherwise Q is chosen. In the following, *Agent* denotes the set containing all possible agents.

To give the Ordered Semantics, we use the labeled transition system $(Agent,\ Label,\ \longrightarrow)$ where $Label = \{\tau\} \cup \{a, \underline{a}, \overline{a}, \neg a \mid a \in Messages\}$ (ranged over by α, β, $\ldots$) is the set of possible labels. The labeled transition relation $\longrightarrow$ is the smallest one satisfying all the axioms and rules in Table 1 plus the symmetric rules of (11) - (14). Due to lack of space, we can not explain here every rule. The interested reader can find the description of all the rules but (9) in [1]. Rule (9) is missing in the original paper since in it no syntactic distinction is made between the inactive parallel composition $\_\|\_$ and the active parallel composition $\_|\_$.

Particular attention should be given to rule (10), which is the one that characterizes the Ordered Semantics: the continuation P and the message $< a >$ reach the tuple space at the same time.

Note that rule (12) uses a negative premise. The authors of [1] claim their operational semantics to be well defined since they can prove that the transition system is strictly stratifiable. To formalize rule (12) in Coq we prefer to substitute the negative premise with an equivalent decidable

auxiliary predicate $NoMessage_a$, which is the smallest predicate such that:
(A) $NoMessage_a(< b >) \quad if a \neq b$
(B) $NoMessage_a(C)$
(C) $\frac{NoMessage(P) \quad NoMessage(Q)}{NoMessage(P|Q)}$

In what follows we are interested only in computations consisting of reduction steps, i.e. the internal derivations that a *stand-alone* agent is able to perform independently of the context. In the Linda Calculus reductions are not only the usual derivations labeled with τ, but also those labeled with $\neg a$. In fact, a derivation $P \xrightarrow{\neg a} P'$ indicates that P can become P' if no tuples $< a >$ are available in the external environment, which is always true for stand-alone processes. Formally we define a new transition $\longrightarrow$ (called *step*) in the following way:

Definition 5. $\longrightarrow = \ (\bigcup_a \xrightarrow{\neg a}) \cup \xrightarrow{\tau}$

Since in process algebras there is no explicit notion of state, the only reasonable form of termination of an agent is the absence of outgoing transitions. Because of the presence of the bang operator, it is senseless to distinguish between properly terminated (i.e. consisting of the parallel composition of agents 0) and deadlocked agents. Thus we say that an agent Q is *stuck* if $Q \not\longrightarrow$ and we say that an agent P terminates if there exist a computation leading to a deadlock, i.e. there exists a stuck agent Q such that $P \longrightarrow^* Q$.

No structural congruence relation is defined in [1] on the agents. Nevertheless many statements given in the paper are true only assuming that processes are identified up to commutativity and associativity of parallel composition and neutrality of 0 with respect to parallel composition. Here we prefer to avoid the definition of a structural congruence relation. Instead in section 5 we will state the soundness theorem using the congruence relation induced by the equivalence relation on RAMs[1].

4.1 The Linda Calculus in Coq

The formalization in Coq of the Linda Calculus syntax and its transition system (where rule (12) is modified as already described) is straightforward. Due to lack of space we do not show here all the constructors of the inductive predicate `OStep` which formalizes the labeled transition:

```
Definition message := nat.

Inductive action : Set :=
   In : message→action
 | Rd : message→action
 | Out : message→action
 | BangIn : message→action.
```

[1] The two congruence relations are provably equal, but we do not provide the formal proof neither here nor in Coq.

```
Inductive program : Set :=
   Dead : program
 | Action : action→program→program
 | IfThenElseAction : action→program→program→program
 | Parallel : program→program→program.

Inductive agent : Set :=
   Message : message→agent
 | Program : program→agent
 | AParallel : agent→agent→agent.

Inductive label : Set :=
   Tau : label
 | LIn : message→label
 | LRead : message→label
 | LOut : message→label
 | LNot : message→label.

(* A step in the ORDERED semantics. *)
Inductive OStep : agent→label→agent→Prop :=
   SMessage :
    ∀m:message.(OStep (Message m) (LOut m) (Program Dead))
 | SOut :
     (* This is the transition that makes the semantics ORDERED *)
     ∀m:message.∀P:program.
      (OStep
       (Program (Action (Out m) P))
       Tau
       (AParallel (Message m) (Program P)))
 | ...
```

The formalization of the Linda Calculus can be found in the Linda.v file, which is about 130 lines long.

5 The Encoding

We start by reviewing the encoding of RAMs into Linda Calculus agents given in [1].

To model a RAM in the Linda Calculus we need both an encoding for the configurations and one for the programs. The program counter in the configuration $(p, c_1, \ldots, c_n)$ is represented by the *program counter tuple* $< p >$ while the contents of registers r_l is modeled by c_l occurrences of the tuple $< r_l >$ (for $l = 1 \ldots n$):

$$[\![(p, c_1, \ldots, c_n)]\!] \stackrel{\text{def}}{=} < p > | \underbrace{< r_1 > | \ldots | < r_1 >}_{c_1 \text{ times}} | \ldots | \underbrace{< r_n > | \ldots | < r_n >}_{c_n \text{ times}}$$

To model the program R composed of the sequence of instructions $i_1 \dots i_m$ we consider the parallel composition of m agents, each one modeling an instruction:

$$
\begin{array}{ll}
[\![R]\!] & \stackrel{\text{def}}{=} [\![i_1]\!]^1 | \dots | [\![i_m]\!]^m \\
[\![\texttt{Succ}(r_j)]\!]^i & \stackrel{\text{def}}{=} !in(p_i).out(r_j).out(p_{i+1}) \\
[\![\texttt{DecJump}(r_j, s)]\!]^i & \stackrel{\text{def}}{=} !in(p_i).inp(r_j)?out(p_{i+1})\_out(p_s)
\end{array}
$$

Each RAM instruction is realized by means of three internal steps: the first one ($!in(p_i)$) consumes the program counter; the second update/tests the contents of the register; the third one introduces the new program counter. The instruction is never consumed because of the replication operator.

Finally, the agent modeling the program R that starts its computation from the configuration $(1, c_1, \dots, c_n)$ is $[\![R]\!]|[\![(1, c_1, \dots, c_n)]\!]$.

The part of [1] that describes the encoding ends with the following theorem:

Theorem 1. *Let R be a RAM program, then:*

- Soundness: *if $[\![R]\!]|[\![(p, c_1, \dots, c_n)]\!] \longrightarrow^3 Q$ then there exists a unique configuration $(p', c'_1 \dots, c'_n)$ such that:*
  $Q = [\![R]\!]|[\![(p', c'_1, \dots, c'_n)]\!]$ and $(p, c_1, \dots, c_n) \longrightarrow_R (p', c'_1, \dots, c'_n)$
- Completeness: *if $(p, c_1, \dots, c_n) \longrightarrow_R (p', c'_1, \dots, c'_n)$ then also*
  $[\![R]\!]|[\![(p, c_1, \dots, c_n)]\!] \longrightarrow^3 [\![R]\!]|[\![(p', c'_1, \dots, c'_n)]\!]$

Proof. By cases on the possible instructions that can be activated. In the proof of soundness we use the fact that the program counter tuple $< p_i >$ in the term $[\![R]\!]|[\![(p_i, c_1, \dots, c_n)]\!]$ ensures that only the agent corresponding to the i^{th} instruction can move. ■

Note in the soundness statement the usage of the equality $=$ between agents. That equality should be intended up to the undefined structural congruence rule.

5.1 The Encoding in Coq

As already suggested in Sect. 3, we decide to encode both register indexes (i.e. numbers in the interval $[0 \dots n)$) and program counter values (i.e. unbounded integers) as natural numbers: a register index i is mapped to i; a program counter value p is mapped to $n + p$.

The encoding relation $[\![\_]\!]$ is straightforwardly formalized in Coq as a set of (recursive) definitions. As an example, we show here only the two functions that map the RAM registers into the corresponding agent:

```
Fixpoint agent_of_register [r: register ; v:nat] : agent :=
 Cases v of
   O ⇒ (Program Dead)
 | (S n) ⇒ (AParallel (Message r) (agent_of_register r n))
 end.
```

```
Fixpoint agent_of_registers [f:register→nat ; n:nat] : agent :=
 Cases n of
   O ⇒ (Program Dead)
 | (S m) ⇒
    (AParallel
     (agent_of_register m (f m))
     (agent_of_registers f m))
 end.
```

Technical induction/recursion principles in the spirit of [7] are sometimes provided to reason on the recursive definitions. Example:

```
Theorem agent_of_registers_ind :
 ∀f:(register→nat).∀registers_no:nat.∀P: agent→Prop.
  (P (Program Dead))→
  (∀r:nat.
   (r<registers_no)→
    (P (agent_of_registers f r))→
     (P (AParallel
          (agent_of_register r (f r)) (agent_of_registers f r))))→
   (P (agent_of_registers f registers_no)).
```

The functions that define the encoding can be found in the file `RAMs_to_Linda.v`, which is only 84 lines long.

We are now left with the formalization of the soundness and completeness statements and their proof.

5.2 The Soundness Proof in Coq

Coq is based on a constructive logic and allows automatic code extraction from constructive proofs. In particular, since we decide to stick ourselves to constructive rules only, we can try to automatically obtain code for:

– Mapping RAMs into Linda agents.
– Recognizing Linda agents that are encoding of a RAM.
– Mapping Linda agents to the RAMs they encode.
– Executing RAMs and Linda agents.

Of course, as we will see later, the price to pay is not neglectable.

Before addressing soundness, we need to characterize Linda agents that are encodings of RAMs. Let's fix two positive integer numbers N and M. A Linda agent is the encoding of a RAM with N available registers and whose program has length M iff:

– the agent is made of parallel compositions of messages $< m >$ (representing either registers or program counters), dead processes 0 and processes which are images of a `Succ` or a `DecJump` instruction (i.e. have either the shape

$!in(p_1).out(r).out(p_2)$ or the shape $!in(p_1).?inp(r).out(p_2)\_out(p_3)$ where $r \in [0, M)$ and $p_i \geq M$ for each i
- there is exactly one message $< m >$ such that $m \geq N$ (which is the encoding of the PC $m - N$)
- for every $m \in [M, \ldots, M + N)$ there is exactly one sub-process which is the image of an instruction and that is guarded by $!in(m)$. The process is the encoding of the $(m - M)$-th instruction
- for every $m \geq M + N$ there is no sub-process which is the image of an instruction and that is guarded by $!in(m)$

Formally, we define in Coq the following (decidable) inductive predicates in Prop:

```
Inductive is_RAM_instruction [registers_no : nat ; pc : nat] :
 program→Prop

Inductive is_RAM_image_no_pc [registers_no : nat] : agent→Prop

Inductive is_RAM_image_pc [registers_no : nat] : agent→Prop

Inductive instruction_does_not_occur [registers_no:nat; n:nat] :
 agent→Prop

Inductive instruction_occurs_once [registers_no:nat ; n:nat] :
 agent→Prop

Inductive is_RAM_image
 [registers_no : nat ; program_length : nat ; P:agent] : Prop
:=
 Build_is_RAM_image:
 (is_RAM_image_pc registers_no P)→
 (∀n:nat.(n<program_length)→
   (instruction_occurs_once registers_no (plus registers_no n) P))→
 (∀n:nat.(n≥program_length)→
   (instruction_does_not_occur registers_no (plus registers_no n) P))→
 (is_RAM_image registers_no program_length P).
```

Note that several of these predicates are used to characterize only sub-processes of RAM images: for example `is_RAM_image_no_pc` is satisfied by RAM image fragments where all the messages are in the $[0, n)$ interval (i.e. are registers).

We also need to provide some more predicates to identify the transient states that are assumed by the Linda agent during the two intermediate steps of the $\longrightarrow^3$ transition:

```
Inductive is_RAM_image_succ [registers_no:nat ; r:register ; pc:nat]:
 agent→Prop
```

```
Inductive is_RAM_image_decjump
 [registers_no:nat ; r:register ; pc':nat ; pc:nat] : agent→Prop

Inductive is_RAM_image_out_pc [registers_no:nat ; pc:nat] :
 agent→Prop

Inductive instruction_does_not_occur2 [registers_no:nat; n:nat] :
 agent→Prop

Inductive instruction_does_not_occur3 [registers_no:nat; n:nat] :
 agent→Prop
```

The first predicate is used to check that no PC message appears in the agent and that the "body" of the `Succ` instruction does (i.e. there is exactly one sub-process $out(r).out(pc)$); the second predicate does the same for the `DecJump` instruction; the third predicate checks that no PC message appears in the agent and that there is exactly one sub-process $out(pc)$; the last two predicates are almost equivalent to the `instruction_does_not_occur` predicate, but are used to state that no instruction guarded by $!in(n)$ is present in agents satisfying respectively the `is_RAM_image_succ`/`is_RAM_image_decjump` predicates or the `is_RAM_image_out_pc` predicate.

Since all the preceding predicates are given in `Prop`[2], the Coq system is able to automatically generate only the corresponding induction and inversion principles over `Prop` (i.e. those where the property P to prove is not computationally relevant and lives in `Prop`). Nevertheless, since all of them are decidable, it is possible to manually define also the corresponding recursion principles (which are extremely useful when proving constructive existential quantifications). In particular, for each inductive predicate we prove a subset of the following lemmas:

1. The decidability proof
2. The corresponding recursion principles (which are proved by induction on the object of the property and using its decidability lemma and the decidability property of all the other properties used in the type of the constructors)
3. An inversion principle in `Set` (i.e. over computational properties)
4. A dependent version of the induction principle

[2] `Prop` is the sort of all the types whose terms have no computational content; subterms of sort `Prop` are removed during code extraction. The propositions whose type is `Prop` can also be proved classically without interfering with the extraction machinery. `Set`, instead, is the sort of the types whose terms have computational content; subterms of sort `Set` are kept during code extraction. Thus any subterm of sort `Set` can not perform any elimination step over a term of sort `Prop`, since that term will not be available in the extracted code.

The defined induction/recursion principles can be easily exploited to prove by induction static properties of RAM images. For example, we use them to prove that no agent satisfies both `is_RAM_image_no_pc` and `is_RAM_image_pc`.

The predicate `is_RAM_image_pc` is particularly important, since it guarantees the existence of just one program counter message in the agent. Later on we will have to prove that an agent satisfying this predicate can make a τ move unless the corresponding RAM is stuck. The τ move will be obtained when the program counter message will be consumed by the program that encodes the corresponding instruction, i.e. the program counter message will perform a $\xrightarrow{\overline{p}}$ move and the instruction program a $\xrightarrow{p}$ move where p is the value of the program counter message. To be able to give a constructive proof of this fact, we need to provide a computable function (in Set) that, given a process that satisfies `is_RAM_image_pc`, gives back its program counter:

```
Definition pc_of_RAM_image_pc :
 Πregisters_no:nat. ΠP:agent.
  (is_RAM_image_pc registers_no P) → message.
```

The definition must be given carefully to ensure its proof-irrelevance in the third argument:

```
Lemma pc_of_RAM_image_pc_proof_irrelevant :
 ∀registers_no:nat. ∀P:agent.
  ∀P0,P1:(is_RAM_image_pc registers_no P).
   (pc_of_RAM_image_pc registers_no P P0) =
   (pc_of_RAM_image_pc registers_no P P1).
```

We must also prove several technical lemmas ensuring that the definition is compositional, i.e. that the value of the program counter of an agent which is a parallel composition is the same value computed on the branch which actually holds the program counter. E.g.:

```
Lemma pc_of_AParallel_equal_pc1 :
 ∀registers_no:nat. ∀P,Q:agent.
  ∀HPQ:(is_RAM_image_pc registers_no (AParallel P Q)).
   ∀HP:(is_RAM_image_pc registers_no P).
    (pc_of_RAM_image_pc ? ? HPQ) = (pc_of_RAM_image_pc ? ? HP).
```

Once we have characterized all the possible states a RAM image can assume, we need to start describing the dynamic properties of the system. In particular, for every possible label and for every possible RAM image state, we need to study which are the possible moves and their outcome. As an example, we show here a very small subset of the main lemmas we need to prove[3]:

[3] The lemmas marked with a $*$ are the one with a computational content.

an agent P that satisfies `is_RAM_image_no_pc`

1. does not perform any $\xrightarrow{\neg a}$ move
2. does not perform any $\xrightarrow{\underline{a}}$ move
3. can perform a $\xrightarrow{a}$ move only if a is a program counter (i.e. $a \geq n$)
4. can perform a $\xrightarrow{\overline{a}}$ move only if a is a register (i.e. $a < n$)
5. if $P \xrightarrow{\overline{a}} P'$ then P' satisfies `is_RAM_image_no_pc`

an agent P that satisfies `is_RAM_image_pc`

1. does not perform any $\xrightarrow{\neg a}$ move
2. does not perform any $\xrightarrow{\underline{a}}$ move
3. can perform a $\xrightarrow{a}$ move only if a is a program counter (i.e. $a \geq n$)
4. $*$ $P \xrightarrow{\overline{p}} P'$ where p is the PC computed by `pc_of_RAM_image_pc`
5. if $P \xrightarrow{\overline{a}} P'$ then P' satisfies `is_RAM_image_no_pc`
6. the $\xrightarrow{\overline{a}}$ move is deterministic both in a and in the produced agent

an agent P that satisfies `instruction_does_not_occur`

1. can not perform a $\xrightarrow{a}$ move on the instruction that does not occur

an agent P that satisfies `instruction_occurs_once`

1. $*$ performs a $\xrightarrow{r}$ move on the instruction guarded by $!in(r)$ that occurs just once
2. the $\xrightarrow{a}$ move is deterministic

an agent P that satisfies `is_RAM_decjump`

1. does not perform any $\xrightarrow{\underline{a}}$ move
2. can perform a $\xrightarrow{a}$ move only if a is a program counter (i.e. $a \geq n$) or if a is the register to be decremented
3. can perform a $\xrightarrow{\overline{a}}$ move only if a is a register (i.e. $a < n$)

...

Proving all these statements without any form of automation turned out to be extremely annoying, since for each proof we need to consider many cases, most of them absurd. Moreover many of the proofs are very similar, since (1) the different predicates just state slightly different properties over similar classes of processes and (2) many cases are due to Linda process constructors or to Linda transitions that are symmetric cases of already considered situations. For example, let's consider a typical proof fragment:

Proof. Given two agents Q and Q' such that `is_RAM_image_pc`(Q) and $Q \xrightarrow{\tau} Q'$, we need to prove a certain property $\mathbf{P}(Q)$. We proceed by induction on the proof of the hypothesis `is_RAM_image_pc`(Q). The two inductive cases are symmetric. The first one is:

Case AParallel1: let Q_1, Q_2 be two agents such that $\texttt{is\_RAM\_image\_pc}(Q_1)$, $\texttt{is\_RAM\_image\_no\_pc}(Q_2)$ and $Q_1|Q_2 \xrightarrow{\tau} Q'$. We need to prove $\mathbf{P}(Q)$. We proceed by cases on the definition of $\xrightarrow{\tau}$. Four out of six cases are absurd:

Case Tau-Right: $Q_2 \xrightarrow{\tau} Q_2'$. Absurd because every process P such that $\texttt{is\_RAM\_image\_no\_pc}(P)$ is stuck.

Case In-Out: $Q_1 \xrightarrow{r} Q_1'$ and $Q_2 \xrightarrow{\overline{r}} Q_2'$. Absurd since $r < n$ (because Q_2 has no program counter) and $r \geq n$ since Q_1 is a RAM image and the only sub-processes that starts with an $rd(\_)$ guard are the instructions which expect to read a program counter message.

Case Read-Out: $Q_1 \xrightarrow{\underline{r}} Q_1'$ and $Q_2 \xrightarrow{\overline{r}} Q_2'$. Absurd since a RAM image can not move with a $\xrightarrow{\underline{r}}$ move.

Case Out-Read: Dual of the previous case.

There are several alternative ways to reduce the size of the proofs and the time spent to develop them:

- Register all the lemmas in the database used by the `Auto` tactic of Coq to try to automatically prove the theorems.
- Develop ad-hoc tactics able to prune out all the typical absurd cases.
- Develop a whole library of elimination principles used to reason on the *dynamic* properties of processes.

The first solution is not very effective and leads to very poor performances during the proof-searching/compilation phase. In fact, just expanding the few `Auto` tactics we were using to the script they were producing, we cut down the overall compilation time from 2m30s to less than 2m. Moreover, as for the second solution, only the size of the script is reduced, and not the size of the generated proof-object (lambda-term). Big proof-objects make type-checking slow, are unmanageable when exported to libraries of mathematical knowledge [8], are difficult to data-mine and to render [9]. Moreover, ad-hoc tactics are a form of *implicit* knowledge that can not be easily shared with other proof assistants, nor presented to the users.

The third solution is the one we tried to pursue[4]. The idea is to introduce one elimination principle à la McBride [4] for each combination of moves and preconditions on the agents that perform the move. In appendix A we show the thesis of the elimination principle over the dynamic behavior of a process P such that $\texttt{is\_RAM\_image\_pc}(P)$, $\texttt{instruction\_occurs\_once}_{pc}(P)$ and $P \xrightarrow{\tau} P'$. The elimination principle can be exploited to enormously reduce the size of both the scripts and the proof-objects of, for example, proof of reduction invariants.

A typical case of a statement whose proof is a simple application of one of these elimination principles[5] is:

[4] So far only a few elimination principles have been introduced. We plan to give all the needed principles and re-prove all the theorems using them.

[5] The one describing the dynamic behavior of RAM images where a `DecJump` instruction is under execution.

```
Lemma is_RAM_image_decjump_Tau_message_multiplicity_eq:
 ∀registers_no:nat. ∀r,m:register. ∀pc,pc':nat. ∀P,P':agent.
  r≠m→
   (is_RAM_image_decjump registers_no r pc' pc P)→
    (OStep P Tau P')→
     (message_multiplicity m P)=(message_multiplicity m P').
```

Once that we have characterized all the possible transitions that RAM images are subject to in each of their states, we can state the completeness axiom and put our attention on the soundness proofs:

```
Axiom RAMs_to_Linda_completeness:
 (r:RAM ; c,c':configuration)
 (RAMs.step r c c')→
  (step3
   (agent_of_RAM_and_configuration r c)
   (agent_of_RAM_and_configuration r c')).

Theorem RAMs_to_Linda_soundness:
∀r:RAM. ∀c:configuration. ∀P':agent.
 ∀H:(step3 (agent_of_RAM_and_configuration r c) P').
  (Σc':configuration |
   (ΣH0:(RAMs.step r c c')
    let Hc be (agent_of_RAM_and_configuration_is_RAM_image r c) in
     let Hc' be (is_RAM_image_step3_is_RAM_image ? ? ? ? Hc H) in
      let Hc'' be (agent_of_RAM_and_configuration_is_RAM_image r c') in
       (RAM_image_equivalence' ? ? P'
        (agent_of_RAM_and_configuration r c') Hc' Hc''))).
```

Note that, to state the soundness theorem, we need to provide another set of lemmas that grants that

- the encoding of a RAM satisfies the predicate `is_RAM_image` (Lemma `agent_of_RAM_and_configuration_is_RAM_image`). The proof is quite straightforward.
- after three steps a RAM image becomes again a RAM image (Lemma `is_RAM_image_step3_is_RAM_image`). The proof is very complex and relies on all the previous lemmas.

Many of the difficulties met in the second proof derive from the combination of dependent types (used all over the theory) with proof-irrelevant definitions (the function to extract the program counter from a RAM image). In particular we were forced to introduce the following axiom[6].

[6] The axiom is surely related to the axiom used to derive Leibniz equality from John-Major equality [4]. As a future work, it should be possible to show the consistency

```
Axiom pc_of_RAM_image_pc_dependent_proof_irrelevant:
 ∀registers_no:nat. ∀P,P':agent.
  ∀H:(is_RAM_image_pc registers_no P).
   ∀H':(is_RAM_image_pc registers_no P').
    P=P'→
     (pc_of_RAM_image_pc registers_no P H) =
      (pc_of_RAM_image_pc registers_no P' H').
```

Notice that both the compleness axiom and the axiom we have just introduced are non-informative (i.e. they are of sort `Prop`). As a consequence, they do not inhibit code extraction, even if inconsistent axioms can destroy the strong normalization property of the extracted code. Indeed, Coq 7.2 successfully extracts code from all the modules that make up our development.

The proof of the soundness theorem is obtained by the completeness axiom and by two further lemmas which state that:

1. A RAM is stuck iff its RAM image is:
2. The $\longrightarrow^3$ relation is deterministic up to the congruence induced by the equality on RAMs:

   ```
   Theorem step3_almost_deterministic:
    ∀registers_no:nat. ∀program_length:nat. ∀P,P',P'':agent.
     (is_RAM_image registers_no program_length P)→
      (step3 P P')→(step3 P P'')→
       ∀H:(is_RAM_image registers_no program_length P').
        ∀H':(is_RAM_image registers_no program_length P'').
         (RAM_image_equivalence' registers_no program_length P' P'' H H').
   ```

The two lemmas are proved combining all the previously proved lemmas. A key observation to prove the second lemma is that the transition relation of RAM images is *almost* deterministic: the only situation where more than one transition is possible is the execution of the $\texttt{DecJump}(r,s)$ instruction, that can remove from the tuple space any instance of the $< r >$ message in case the value of the register r (in the corresponding RAM) is greater than 1.

Once the lemmas have been proved, the proof becomes almost straightforward. Informally, if the agent that encodes the RAM r and the configuraiont c can move in a new agent P', than the couple (r,c) must move in a new state (r,c') (because of the first lemmas); the completeness property grants that the image P of (r,c) moves into a new agent P''; since the second lemma states that the reduction $\longrightarrow^3$ is deterministic (up to the congruence induced by the equality on RAMs), we must conclude that $P' = P''$, ending the proof.

of our axiom by adapting one of the modes that validates the John-Major equality axiom.

All the lemmas and proof can be found into the two files `RAMs_to_Linda_proofs.v` and `RAMs_to_Linda_proofs1.v`, which are currently about 8700 lines long and require some minutes to be compiled.

6 Conclusions

In this paper we present the formalizations in Coq of both Random Access Machines (RAMs) and a Turing complete version of a process algebra built around the Linda coordination primitives. We also give an encoding of RAMs into the process algebra and a constructive proof of the soundness of the encoding.

The formalizations of the two calculi as well as the formalization of the encoding are almost straightforward. The soundness proof, instead, turned out to be extremely long and complex. The deep reasons of the complexity of some parts of the proof can be traced back to the constructive nature of the proof and the heavy usage of functions having both a computational content and proof-irrelevant arguments (i.e. functions of the form $\Pi x : A.(P\ x) \rightarrow B\ :\ Set$). More concretely, though, the proof required so much work because of lack of automation in the Coq system: in particular Coq provides only a very limited support for the generation of inversion/induction principles over *decidable* predicates and no automation at all for the handling of “similar” or “symmetric” cases.

Finally, we strongly believe that both the size of the proof script and of the proof-objects can be highly reduced introducing many more elimination and inversion principles to characterize the dynamic behavior of the processes. The generation of these principles, though, should be automatized, since even guessing their precise statement is a non-trivial task.

The difficulties we have met and the solutions proposed seem to be unrelated from the particular proof we are examining. Thus further work could be spent in trying to generalize the technique and apply it to other proofs about the mutual encoding of formalisms. Automatizing part of the proofs would be very promising. As a matter of facts, though, it is evident that without any additional automation, the Coq system is not very effective in dealing with this kind of proofs: even for relatively trivial facts, the time spent in the development of the boring details of a constructive and fully formalized proof largely overcomes the time required by the interesting steps.

In the next few months we are going to finish the whole development giving also the completeness proof and extracting the code to read-back a RAM from its Linda encoding.

A long term goal would be to define inside the RAM formalism its own interpreter and then proving the undecidability of the halting problem. This would really pave the way to far more interesting results, as the proof that the two versions of the Linda Calculus (with Ordered and Unordered Semantics) are not equivalent.

References

1. N. Busi, R. Gorrieri and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156(1/2):90-121, 2000.
2. The Coq proof-assistant:
`http://coq.inria.fr/`
3. M. Hofmann. *Extensional concepts in intensional type theory.* Ph.D. thesis, University of Edinburgh, July 1995.
4. C. McBride. *Dependently Typed Functional Programs and their Proofs.* Ph.D. thesis, University of Edinburgh, 2000.
5. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, 1985.
6. D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97-102, 1992.
7. G. Barthe and P. Courtieu. Efficient Reasoning about executable specifications in Coq. In V. Carreño, C. Munõz and S. Tahar editors, *Theorem Proving in Higher Order Logics (TPHOLS)*, volume 2410 of *LNCS*, pages 31-46, Springer Verlag, 2001.
8. A. Asperti, F. Guidi, L.Padovani, C. Sacerdoti Coen and I. Schena. Mathematical Knowledge Management in HELM. In *On-Line Proceedings of the First International Workshop on Mathematical Knowledge Management (MKM2001)*, RISC-Linz, Austria, September 2001.
9. A. Asperti, F. Guidi, L.Padovani, C. Sacerdoti Coen and I. Schena. XML, Stylesheets and the re-mathematization of Formal Content. In *On-Line Proceedings of EXTREME 2001.*
10. R. Milner. The Polyadic π-Calculus: A Tutorial Technical Report, Department of Computer Science, University of Edinburgh, ECS-LFCS-91-180, October 1991.
11. M. L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Englewood Cliffs, 1967.
12. J. C. Shepherdson and J. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217-255, 1963.
13. N. Busi, R. Gorrieri and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. Theoretical Computer Science, 192(2):167-199, 1998.
14. N. Busi and G. Zavattaro. On the Expressiveness of Movement in Pure Mobile Ambients. In *Prof. of Workshop on Peer-to-Peer Computing*, LNCS 2376, Springer-Verlag, 2002.

A One Example of Elimination Principle Characterizing a Dynamic Behaviour

```
Lemma is_RAM_image_pc_instruction_occurs_once_Tau_ind:
∀registers_no:nat. ∀P:(agent→agent→Prop).
 (∀P0,P',Q:agent.
  ∀H:(is_RAM_image_pc registers_no P0).
   (is_RAM_image_no_pc registers_no Q)→
   let pc be (pc_of_RAM_image_pc ? ? H) in
    (* This hypothesis can be strengthened *)
```

```
   (instruction_occurs_once registers_no pc (AParallel P0 Q))→
    (P P0 P')→(P (AParallel P0 Q) (AParallel P' Q)))→
(∀P0,Q,Q':agent.
 (is_RAM_image_no_pc registers_no P0)→
  (is_RAM_image_pc registers_no Q)→
   let pc be (pc_of_RAM_image_pc ? ? H) in
   (* This hypothesis can be strengthened *)
   (instruction_occurs_once registers_no pc (AParallel P0 Q))→
    (P Q Q')→(P (AParallel P0 Q) (AParallel P0 Q'))→
(∀P0,P',Q,Q':agent. ∀r:register.
 ∀H:(is_RAM_image_pc registers_no P0).
  (is_RAM_image_no_pc registers_no Q)→
   let pc be (pc_of_RAM_image_pc ? ? H) in
   (* This hypothesis can be strengthened *)
   (instruction_occurs_once registers_no pc (AParallel P0 Q))→
    (OStep P0 (LOut r) P')→
     (OStep Q (LIn r) Q')→
      (P (AParallel P0 Q) (AParallel P' Q')))→
(∀P0,P',Q,Q':agent. ∀r:register.
 (is_RAM_image_no_pc registers_no P0)→
  ∀H:(is_RAM_image_pc registers_no Q).
   let pc be (pc_of_RAM_image_pc ? ? H) in
   (* This hypothesis can be strengthened *)
   (instruction_occurs_once registers_no pc (AParallel P0 Q))→
    (OStep P0 (LIn r) P')→
     (OStep Q (LOut r) Q')→
      (P (AParallel P0 Q) (AParallel P' Q')))→
∀a,a':agent.
 ∀H:(is_RAM_image_pc registers_no a).
  let pc be (pc_of_RAM_image_pc ? ? H) in
  (instruction_occurs_once registers_no pc a)→
   (OStep a Tau a')→
    (P a a').
```