HAL
open science

Actes de la conférence JFLA 2009 (Vingtièmes Journées Francophones des Langages Applicatifs)

Alan Schmitt

To cite this version:

Alan Schmitt. Actes de la conférence JFLA 2009 (Vingtièmes Journées Francophones des Langages Applicatifs). INRIA. INRIA, pp.174, 2009. inria-00362717

HAL Id: inria-00362717

https://inria.hal.science/inria-00362717v1

Submitted on 26 Feb 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ACTES DE LA CONFERENCE

Journées Francophones des Langages Applicatifs

Responsable scientifique : Alan Schmitt

JFLA2009

31 janvier - 3 février 2009
Saint-Quentin-sur-Isère

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

Avant-propos

Alan Schmitt[1] & Micaela Mayero[2]

1: LIG, INRIA Grenoble - Rhône-Alpes
2: LIPN, Université Paris 13

> Les jambes de vingt ans sont faites pour aller au bout du monde.
>
> Christian Bobin

Pour la vingtième année consécutive, les Journées Francophones des Langages Applications sont l'occasion de se retrouver dans un cadre agréable et propice aux échanges conviviaux. Cette année, c'est à Saint-Quentin sur Isère, près de Grenoble, que nous nous réunissons, maintenant la tradition de l'alternance mer-montagne.

Les neuf articles choisis par le comité de programme reflètent bien la diversité de notre communauté et les avancés tant du point de vue de l'application de langages fonctionnels que de la conception et de l'utilisation d'assistants à la preuve. Nous avons souhaité également inclure des articles plus proches de tutoriels ou de retours d'expérience, ceux-ci étant particulièrement adaptés au cadre pédagogique des Journées.

Deux orateurs nous ont fait l'honneur d'accepter notre invitation. L'exposé de Vincent Balat, de l'université Paris 7, intitulé « Ocsigen : approche fonctionnelle typée de la programmation Web » illustre l'utilisation croissante de langages applicatifs dans des milieux inattendus. L'exposé de Bruno Barras, de Trusted Labs, intitulé « Faut-il avoir peur de sa carte SIM ? » présente l'application d'assistants à la preuve dans la modélisation de cartes à puces.

Pour la quatrième année consécutive, deux sessions d'une demi-journée chacune sont consacrées à des cours. Le premier porte sur la modélisation de la linguistique (par Gérard Huet, de l'INRIA Paris - Rocquencourt) et le deuxième sur les bibliothèques Coq utilisées dans la preuve récente du théorème des quatre couleurs (par Assia Mahboubi, de l'INRIA Saclay - Île-de-France).

Nous remercions chaleureusement les membres du comité de programme : Boutheina Chetali, Sylvain Conchon, David Delahaye, Hugo Herbelin, Didier Le Botlan, Jean-Vincent Loddo, Alexandre Miquel et Davide Sangiorgi.

Nous remercions également Sylvie Boldo, Thomas Braibant, Pierre Courtieu, Catherine Dubois, Michaël Lienhardt, Christine Paulin et Christophe Tollu qui ont participé à la relecture des articles soumis.

Les JFLA existent grâce au support constant de l'INRIA et nous voulons chaleureusement remercier Sophie Azzaro, Danièle Herzog et Laëtitia Libralato, du Bureau des Cours et Colloques de l'INRIA Grenoble - Rhône-Alpes pour la prise en charge de l'organisation matérielle de ces Journées. Les JLFA ne seraient rien sans l'investissement de Pierre Weis : non seulement il en gère le site web et coordonne de main de maître toutes les étapes de préparation des Journées, mais il a également accepté de nous raconter « Vingt années de JFLA ». Qu'il en soit ici très profondément remercié !

Table des matières

Calculs applicatifs de machines relationnelles (cours) 1
Présentation de SSReflect (cours) 21
Ocsigen : approche fonctionnelle typée de la programmation Web (conférence invitée) . . . 23
Qui sème la fonction, récolte le tuyau typé 25
Foncteurs impératifs et composés : la notion de projets dans Frama-C 37
Vers une programmation fonctionnelle en appel par valeur sur systèmes multi-coeurs : évaluation asynchrone et ramasse-miettes parallèle 55
Vérification d'invariants pour des systèmes spécifiés en logique de réécriture 73
Un modèle de l'assistant à la preuve : PAF ! 89
Extraction certifiée dans Coq-en-Coq 105
Abstraction d'horloges dans les systèmes synchrones flot de données 119
Faut-il avoir peur de sa carte SIM ? (conférence invitée) 135
Fouille au code OCaml par analyse de dépendances 137
Faire bonne figure avec MLPOST 155

Calculs applicatifs de machines relationnelles

Gérard Huet & Benoît Razet

Centre INRIA de Paris-Rocquencourt
Gerard.Huet@inria.fr
Benoit.Razet@inria.fr

Résumé

Ce texte est un support du cours "Automates, transducteurs et machines d'Eilenberg applicatives dans la boîte à outils Zen — Applications au traitement de la langue" de Gérard Huet aux vingtièmes journées francophones des langages applicatifs (JFLA2009) à Saint-Quentin sur Isère. Le cours comporte trois volets. Tout d'abord, les structures de base de la boîte à outils Zen de traitement de données linguistiques sont présentées. Cette bibliothèque de modules Objective Caml, disponible librement sous forme source avec licence LGPL en `http://sanskrit.inria.fr/ZEN/`, est commentée en style Ocamlweb dans le document `http://sanskrit.inria.fr/ZEN/zen.pdf`. Le cours comporte ensuite une démonstration de plate-forme de traitement du sanskrit, utilisant cette bibliothèque pour les niveaux phonologiques et morphologiques, pour la représentation des lexiques et des transducteurs, enfin pour la lemmatisation, la segmentation, l'étiquetage et l'analyse superficielle. Cette plate-forme, entièrement implémentée en OCaml, est utilisable comme service Web à l'URL `http://sanskrit.inria.fr/`. Les tables morphologiques du sanskrit qu'elle construit sont disponibles librement sous forme XML/DTD avec licence LGPLLR. en `http://sanskrit.inria.fr/DATA/XML/`.

Le cours montre enfin comment les divers processus à l'œuvre dans cette application sont des cas particuliers de *machines d'Eilenberg finies* au sens de Benoît Razet. La méthodologie peut se comprendre comme le cas fini d'un modèle de calcul non déterministe très général, faisant communiquer des machines relationnelles exécutant des actions non déterministes. Ce calcul est exprimé fonctionnellement par un calcul progressif de flots de solutions, géré par un moniteur séquentiel appelé "moteur réactif" et paramétré par une stratégie de recherche. L'article en anglais qui suit présente succintement cette méthodologie et donne quelques références complémentaires.

Ce matériau a déjà été présenté dans ses grandes lignes par Gérard Huet au cours "Structures Informatiques et Logiques pour la Modélisation Linguistique" (MPRI 2-27-1) à Paris à l'automne 2008, ainsi qu'au tutoriel "Eilenberg machines, the Zen toolkit, and applications to Sanskrit Computational Linguistics" de Gérard Huet et Benoît Razet au congrès ICON-2008 (6th International Conference On Natural Language Processing) à Pune, Inde, en décembre 2008. La thèse de Benoît Razet (à paraître en 2009) développe complètement ce modèle de calcul, donne des extensions équitables au cas infini, décrit complètement la compilation du contrôle à partir d'expressions régulières, et valide formellement les propriétés (correction et complétude) du moteur réactif par une certification dans le système de preuves Coq.

Computing with Relational Machines

Abstract

We give a quick presentation of the X-machines of Eilenberg, a generalisation of finite state automata suitable for general non-deterministic computation. Such machines complement an automaton, seen as its control component, with a computation component over a data domain specified as an action algebra. Actions are interpreted as binary relations over the data domain, structured by regular expression operations. We show various strategies for the sequential simulation of our relational machines, using variants of the *reaction engine*. In a particular case of *finite machines*, we show that bottom-up search yields an efficient complete simulator.

Relational machines may be composed in a modular fashion, since atomic actions of one machine may be mapped to the characteristic relation of other relational machines acting as its parameters.

The control components of machines is compiled from regular expressions. Several such translations have been proposed in the literature, that we briefly survey.

Our view of machines is completely applicative. They may be defined constructively in type theory, where the correctness of their simulation may be formally checked. From formal proofs in the Coq proof assistant, efficient functional programs in the Objective Caml programming language may be mechanically extracted.

Most of this material is extracted from the (forthcoming) Ph.D. thesis of Benoît Razet.

1. Machines

1.1. Relational machines

We shall define a notion of abstract machine inspired from the work of Eilenberg (X-machines, presented in [8]). Our machines are non-deterministic in nature. They comprise a *control component*, similar to the transitions state diagram of a (non-deterministic) automaton. These transitions are labeled by action generators. Action expressions over free generators, generalizing regular expressions from the theory of languages, provide a specification language for the control component of machines. A program, or action expression, compiles into control components according to various translations. Control components in their turn may compile further into transition matrices or other representations.

Our machines also comprise a *data component*, endowed with a relational semantics. That is, we interpret action generators by semantic attachments to binary relations over the data domain. These relations are themselves represented as functions from data elements to streams of data elements. This applicative apparatus replaces by clear mathematical notions the imperative components of traditional automata (tapes, reading head, counters, stacks, etc).

We shall now formalise these notions in a way which will exhibit the symmetry between control and data. First of all, we postulate a finite set Σ of parameters standing for the names of the primitive operations of the machine, called *generators*.

For the control component, we postulate a finite set S of states and a *transition relation map* interpreting each generator as a (binary) relation over S. This transition relation interpretation is usually presented currified as a *transition function* δ mapping each state in S to a finite set of pairs (a, q) with a a generator and q a state. This set is implemented as a finite list of such pairs.

Finally, we select in S a set of *initial states* and a set of *accepting states*.

For the data component, we postulate a set D of data values and a *computation relation map* interpreting each generator as a (binary) relation over D. Similarly as for the control component, we

shall currify this relation map as a *computation function* mapping each generator a in Σ to a function $\rho(a)$ in $D \to \wp(D)$. Now the situation is not like for control, since D and thus $\wp(D)$ may be infinite. In order to have a constructive characterization, we shall assume that D is recursively enumerable, and that each $\rho(a)$ maps $d \in D$ to a recursively enumerable subset of $\wp(D)$. We shall represent such subsets as progressively computed streams of values, as we shall explain in due time.

1.2. Progressive relations as streams

We recall that a recursively enumerable subset of ω is the range of a partial recursive function in $\omega \rightharpoonup \omega$, or equivalently it is either empty or the range of a (total) recursive function in $\omega \to \omega$. None of these two definitions is totally satisfying, since in the first definition we may loop on some values of the parameter, obliging us to dovetail the computations in order to obtain a sequence of elements which enumerates completely the set, and in the second we may stutter enumerating the same element in multiple ways. This stuttering cannot be totally eliminated without looping, for instance for finite sets. Furthermore, demanding total functions is a bit illusory. It means either we restrict ourserves to a non Turing-complete algorithmic description language (such as primitive recursive programs), or else we cannot decide the totality of algorithms demanded by the definition.

We shall here assume that our algorithmic description language is ML, i.e. typed lambda-calculus evaluated in call by value with a recursion operator, inductive types and parametric modules. More precisely, we shall present all our algorithms in the Objective Caml implementation.

In this framework we may define computable streams over a parametric datatype `data` as follows:

```
type stream 'data =
 [ Void
 | Stream of 'data and delay 'data
 ]
and delay 'data = unit → stream 'data;
```

This expresses that a stream of data values is either `Void`, representing the empty set, or else a pair `Stream(d,f)` with d of type `data`, and f a frozen stream value, representing the set $\{d\} \cup F$, where F may be computed as the stream $f()$, where $()$ is syntax for the canonical element in type `unit`. Using this inductive parametric datatype, we may now define progressive relations by the following type:

```
type relation 'data = 'data → stream 'data;
```

1.3. Kernel machines

We now have all the ingredients to define the module signature of *kernel machines*:

```
module type EMK = sig
 type generator;
 type data;
 type state;
 value transition: state → list (generator × state);
 value initial: list state;
 value accept: state → bool;
 value semantics : generator → relation data;
end;
```

In the following, we shall continue to use Σ (resp. D, S, δ, ρ) as shorthand for `generator` (resp. `date`, `state`, `transition`, `semantics`). We also write I for `initial` and T for the set of accepting states (for which the predicate `accept` is true).

A machine is like a blackbox, which evolves through series of non-deterministic computation steps. At any point of the computation, its status is characterized by the pair (s, d) of its *current state* $s \in S$ and its *current data value* $d \in D$. Such a pair is called a *cell*.

A computation step issued from cell (s, d) consists in choosing a transition $(a, s') \in \delta(s)$ and a value $d' \in \rho(a)(d)$. If any of these choices fails, because the corresponding set is empty, the machine is said to be *blocked*; otherwise, the computation step succeeds, and the machine has as status the new cell (s', d'). We write $(s, d) \xrightarrow{a} (s', d')$.

A *computation path* consists of such computations steps:

$$(s_0, d_0) \xrightarrow{a_1} (s_1, d_1) \xrightarrow{a_2} (s_2, d_2) ... \xrightarrow{a_n} (s_n, d_n)$$

The computation is said to be *accepting* whenever $s_0 \in I$ and $s_n \in T$, in which case we say that the machine *accepts* input d_0 and *computes* output d_n. Remark that (d_0, d_n) belongs to the graph of the composition of relations labeling the path: $\rho(a_1) \circ \rho(a_2) \circ ...\rho(a_n)$.

We have thus a very general model of relational calculus. Our machines compute relations over the data domain D, and we shall thus speak of *D-machines*. The "machine language" has for instructions the action generators. Actions compose by computation. Furthermore, a high level programming language for relational calculus may be designed as an action calculus. The obvious point of departure for this calculus is to consider regular expressions, in other words the free Kleene algebra generated by the set of generators. We know from automata theory various translations from regular expressions to finite-state automata. Every such translation gives us a compiler of our action algebra into the control components of our machines: S, δ, I and T. The data components, D and ρ, offer a clean mathematical abstraction over the imperative paraphernalia of classical automata: reading heads, tapes, etc. And we get immediately a programming language enriching the machine language of primitive actions by composition, iteration, and choice.

Indeed, a finite automaton over alphabet Σ is readily emulated by the machine with generator set Σ having its state transition graph as its control component, and having for data domain the free monoid of actions Σ^*. Each generator a is interpreted in the semantics as the (functional) relation $\rho(a) = L_a^{-1} =_{def} \{(a \cdot w, w) \mid w \in \Sigma^*\}$ which "reads the input tape". And indeed the language recognized by the automaton is retrieved as the composition of actions along all accepting computations. Here the data computation is merely a trace of the different states of the "input tape".

This example is a simple one, and data computation is deterministic, since $\rho(a)$ is a partial function. We may say that such a machine is "data driven". Control will be deterministic too, provided the underlying automaton is deterministic, since every $\delta(s)$ will then have a unique non-blocking transition. But remark that the same control component may be associated with different semantics. For instance, with $\rho(a) = R_a =_{def} \{(w, w \cdot a) \mid w \in \Gamma^*\}$, the machine will enumerate with its accepting computations the regular language recognized by the automaton.

Let us now turn towards the action calculus.

2. Actions

Actions may be composed. We write $A \cdot B$ for the composition of actions A and B. This corresponds to the composition of the underlying relations.

Actions may be iterated. We write A^+ for the iteration of action A. This corresponds to the transitive closure of the underlying relation. We postulate an identity action 1 corresponding to the underlying identity relation.

Actions may be summed. We write $A + B$ for the sum of actions A and B. This corresponds to the union of the underlying relations. We note A^* for $1 + A^+$. We also postulate an empty action 0.

The algebraic structure of actions is that of a composition monoid:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

$$A \cdot 1 = 1 \cdot A = A$$

and for union, an idempotent abelian monoid:

$$(A + B) + C = A + (B + C)$$

$$A + B = B + A$$

$$A + 0 = 0 + A = A$$

$$A + A = A$$

verifying distributivity:

$$A \cdot (B + C) = A \cdot B + A \cdot C$$

$$(A + B) \cdot C = A \cdot C + B \cdot C$$

$$A \cdot 0 = 0 \cdot A = 0$$

and thus, so far actions form an idempotent semiring. Defining $A \leq B =_{def} A + B = B$, the partial ordering $\leq$ makes the algebra of actions an upper semilattice.

As for iteration (which will be interpreted over relations by transitive-reflexive closure), we follow Pratt [15] in adding implications between actions, in order to get an algebraic variety (as opposed to Kleene algebras, which only form a quasi variety, i.e. need conditional identities for their complete axiomatisation). Thus we postulate $\leftarrow$ and $\rightarrow$, corresponding to relational semi-complements:

$$\rho \rightarrow \sigma = \{(v, w) \mid \forall u \; u\rho v \Rightarrow u\sigma w\}$$

$$\sigma \leftarrow \rho = \{(u, w) \mid \forall v \; w\rho v \Rightarrow u\sigma v\}$$

and we axiomatise actions as *residuation algebras*, following Kozen [14]:

$$A \cdot C \leq B \Leftrightarrow C \leq A \rightarrow B$$

$$C \cdot A \leq B \Leftrightarrow C \leq B \leftarrow A$$

or alternatively we may replace these two equivalences by the following equational axioms:

$$A \cdot (A \rightarrow B) \leq B$$

$$(B \leftarrow A) \cdot A \leq B$$

$$A \rightarrow B \leq A \rightarrow (B + C)$$

$$B \leftarrow A \leq (B + C) \leftarrow A$$

$$A \leq B \rightarrow (B \cdot A)$$

$$A \leq (A \cdot B) \leftarrow B$$

We may now get Pratt's action algebras by axiomatizing iteration as pure induction:

$$1 + A + A^* \cdot A^* \leq A^*$$

$$(A \rightarrow A)^* = A \rightarrow A$$

$$(A \leftarrow A)^* = A \leftarrow A$$

The residuation/implication operations may be seen as the right interpolants to extend conservatively Kleene algebras to the variety of action algebras. Regular expressions and their compilation extend gracefully to action expressions, and the residuation operations correspond to Brzozowki's derivatives.

Furthermore, following Kozen [14], we may wish to enrich our actions with a multiplicative operation $\cap$, corresponding to relation intersection, verifying lower semilattice axioms:

$$(A \cap B) \cap C = A \cap (B \cap C)$$

$$A \cap B = B \cap A$$

$$A \cap A = A$$

and completing to a lattice structure with:

$$A + (A \cap B) = A$$

$$A \cap (A + B) = A$$

obtaining thus Kozen's action lattices, the right structure for matrix computation.

We remark that such structures go in the direction of logical languages, since union, intersection and residuation laws are valid Heyting algebras axioms. We are still far from the complete Boolean algebra structure of relations, though.

3. Behaviour and interfaces

We recall that we defined above the accepting computations of a machine, and for each such computation its compound action, obtained by composing the generating relations of each computation step. Let us call *behaviour* of a machine $\mathcal{M}$ the set of all such compound actions, noted $|\mathcal{M}|$.

Now we define the *characteristic relation* of a machine $\mathcal{M}$ as the union of the semantics of its behaviour:

$$||\mathcal{M}|| = \bigcup_{a \in |\mathcal{M}|} \rho(a)$$

Characteristic relations are the relational interpretation over the data domain D of the action langage recognized by the underlying automaton. They allow us to compose our machines in modular fashion.

3.1. Modular construction of machines

Now that we understand that a D-machine implements a relation over D, we may compose machines vertically, as follows. Let $\mathcal{A}$ be a (non-deterministic) automaton over alphabet Σ, and for every $a \in \Sigma$ let $\mathcal{N}_a$ be a D-machine over some generator set Σ_a. We may now turn $\mathcal{A}$ into a D-machine over generator set Σ by taking $\mathcal{A}$ as its control component, and extending it by a data component having as semantics the function mapping $a \in \Sigma$ to $||\mathcal{N}_a||$.

We may thus construct large machines from smaller ones computing on the same data domain. A typical example of application for computational linguistics is to do morphological treatment (such as segmentation and tagging of some corpus) in a lexicon-directed way. The alphabet Σ defines the lexical categories or parts of speech, each machine $\mathcal{N}_a$ implements access to the lexicon of category a, the automaton $\mathcal{A}$ defines the morphological geometry, and the composite machine $\mathcal{M}$ implements a lexicon-directed parts-of-speech tagger. By appropriate extension of the lexicon machines $\mathcal{N}_a$, morpho-phonemic treatment at the junction of the words may be effected, such as complete sandhi analysis for Sanskrit [11, 12].

3.2. Interfaces

What we described so far is the Eilenberg machine *kernel*, consisting of its control and data elements. We may complete this description by an *interface*, composed of an input domain D_-, an output domain D_+, an input relation ϕ_- and an output relation ϕ_+. A machine $\mathcal{M}$ completed by this interface I defines a relation $\phi(M, I) : D_- \to D_+$ by composition:

$$\phi(M, I) = \phi_- \circ ||\mathcal{M}|| \circ \phi_+$$

4. Finite machines

We shall now present an important special case of machines which exhibit a finite behaviour.

The relation $\rho : D \to D'$ is said to be *locally finite* if for every $d \in D$ the set $\rho(d)$ is finite. The machine $\mathcal{M}$ is said *locally finite* if every relation $\rho(a)$ is locally finite [9]. The machine $\mathcal{M}$ is said *nœtherian* if all its computations are finite in length.

We remark that a machine is nœtherian when its data domain D is a well-founded ordering for the order relation $>$ generated by:

$$d > d' \;\Leftarrow\; \exists a \in \Sigma \; d' \in \rho(a)(d)$$

Indeed, if there existed an infinite computation, there would exist an infinite sub-sequence going through the same state. But the converse is not true, since a machine may terminate for a reason depending of its control.

Finally, we say that a machine is *finite* if it is locally finite and nœtherian.

We say that a machine is *sequential* [8] iff for each cell value (s, d) occurring in a computation there exists at most one computation transition issued from it, i.e. if $\delta(s)$ is a set of pairs $\{(\rho_1, s_1), (\rho_2, s_2), ...(\rho_n, s_n)\}$ such that for at most one $1 \leq k \leq n$ the set $\rho_k(d)$ is non empty, and if such k exists then $\rho_k(d)$ is a singleton. This condition demands that on one hand the transition relation of the underlying automaton is a partial function, that is the automaton must be deterministic, and on the other hand that the relations leading out of a state s be partial functions over the subset of D which is reachable by computation leading to s. We remark that a sequential machine may nevertheless generate several solutions, since a terminal cell is not necessarily blocking further computation.

4.1. Examples

4.1.1. Non deterministic finite automata

Let us consider a non-deterministic automaton $\mathcal{A}$ with parameters (S, I, T, δ). We construct an Eilenberg machine $\mathcal{M}$ solving the word problem for the rational language $|\mathcal{A}|$ recognized by the automaton. $\mathcal{M}$ has Σ for generating set, and it takes $\mathcal{A}$ for its control component. For the data component, we take $D = \Sigma^*$, and the semantics is $\rho(a) = L_a^{-1} =_{def} \{(a \cdot w, w) \mid w \in \Sigma^*\}$, as explained above.

We may check that $\rho(w) = 1$ iff $w \in |\mathcal{A}|$. It is easy to check that $\mathcal{M}$ is finite, since data decreases in length, and semantics is a partial function. When $\mathcal{A}$ is a deterministic automaton, $\mathcal{M}$ is a sequential machine.

Another machine with the same control component may be defined to enumerate all the words in set $|\mathcal{A}|$. In general it will neither be finite, nor sequential.

4.1.2. Rational transducers

Let Σ and Γ be two finite alphabets. A transducer $A : \Sigma \Rightarrow \Gamma$ is similar to a (non-deterministic) automaton, whose transitions are labeled with pairs of words in $D = \Sigma^* \times \Gamma^*$. Let Ω be the (finite) set of labels occurring as labels of the transitions of $\mathcal{A}$. The transition graph of $\mathcal{A}$ may thus be considered as an ordinary non-deterministic automaton over generator alphabet Ω, and constitutes the control component of the machines we shall define to solve various transductions tasks.

We recall that a transducer "reads its input" on an input tape representing a word in Σ^* and "prints its output" on an output tape representing a word in Γ^*. On transition (w, w') it reads off w on the input tape, and if successful appends w' to its output tape. If by a succession of transitions starting from an initial state with input i and empty output it reaches an accepting state with empty input and output o, we say that (i, o) belongs to the *rational relation* in $\Sigma \Rightarrow \Gamma$ recognized by the transducer $\mathcal{A}$, which we shall write $|\mathcal{A}|$. We shall now solve various decision problems on $|\mathcal{A}|$ using machines which use $\mathcal{A}$ for control and D for data, but replace the tapes by various semantic functions:

1. Recognition. Given $(w, w') \in D$, decide whether $(w, w') \in |\mathcal{A}|$.

2. Synthesis. Given $w \in \Sigma^*$, compute its image $|\mathcal{A}|(w) \subset \Gamma^*$.

3. Analysis. Given $w \in \Gamma^*$, compute the inverse image $|\mathcal{A}^{-1}|(w) \subset \Sigma^*$.

Recognition. The semantics ρ is defined by $\rho(\sigma, \gamma) = L_\sigma^{-1} \times L_\gamma^{-1}$. Like for ordinary automata we obtain a finite machine, provided the transducer has no transition labeled (ϵ, ϵ), since at least one of the two lengths decreases. We choose as interface $D_- = \Sigma^* \times \Gamma^*$, $\phi_- = Id_{\Sigma^* \times \Gamma^*}$, $D_+ = 0, 1$, $\phi_+(w, w') = 1$ iff $w = w' = \epsilon$.

Synthesis. The semantics ρ is defined by $\rho(\sigma, \gamma) = L_\sigma^{-1} \times R_\gamma$, with $R_\gamma =_{def} \{(w, w \cdot \gamma) \mid w \in \Gamma^*\}$. We choose as interface $D_- = \Sigma^*$, $\phi_- = \{(w, (w, \epsilon)) \mid w \in \Sigma^*\}$, $D_+ = \Gamma^*$, $\phi_+ = \{(\epsilon, w'), w') \mid w' \in \Gamma^*\}$. We get $|\mathcal{A}| = \phi_- \circ ||\mathcal{M}|| \circ \phi_+$. Such a machine is locally finite, since relations L_σ^{-1} and R_γ are partial functions. However, it may not be nœtherian, since there may exist transitions labeled with actions (ϵ, w). Actually the machine is nœtherian iff cycles of such transitions do not occur, iff the set $|\mathcal{A}|(w)$ is finite for every $w \in \Sigma^*$ [16].

Analysis. Symmetric to synthesis, replacing L_σ^{-1} by R_σ and R_γ by L_γ^{-1}.

4.1.3. Oracle machines

Let D be an arbitrary set, and P an arbitarry predicate over D. We consider the relation ρ over D defined as the restriction of identity to the data elements verifying P: $\rho(d) = \{d\}$ if $P(d)$, $\rho(d) = \emptyset$ otherwise. We define in a canonical way the machine whose control component is the automaton $\mathcal{A}$ with two states $S = \{0, 1\}$, $I = \{0\}$ and $T = \{1\}$, and transition function δ defined by $\delta(0) = \{(\rho, 1)\}$ and $\delta(1) = \emptyset$. This machine is a sequential finite machine, that decides in one computational step whether its input verifies P. Our restriction of Eilenberg machines to computable relations limits such oracles to recursive predicates, but of arbitrary complexity. More generally, our machines recursively enumerate arbitrary recursively enumerable sets, and are therefore Turing complete.

5. Reactive engine

We may simulate the computations of a finite Eilenberg machine by adapting the notion de *reactive engine* of the Zen library [10, 11, 12, 16].

5.1. The depth-first search reactive engine

```
module Engine (Machine: EMK) = struct
open Machine;

type choice = list (generator × state);

(* We stack backtrack choice points in a resumption *)
type backtrack =
 [ React of data and state
 | Choose of data and choice and delay data and state
 ]
and resumption = list backtrack;

(* The 3 internal loops of the reactive engine *)

(* react: data → state → resumption → stream data *)
value rec react d q res =
 let ch = transition q in
 (* we need to compute [choose d ch res] but first
    we deliver data [d] to the stream of solutions when [q] is accepting *)
 if accept q
   then Stream d (fun () → choose d ch res) (* Solution d found *)
   else choose d ch res

(* choose: data → choice → resumption → stream data *)
and choose d ch res =
 match ch with
 [ [] → resume res
 | [ (g, q') :: rest ] → match semantics g d with
             [ Void → choose d rest res
             | Stream d' del → react d' q' [ Choose d rest del q' :: res ]
             ]
 ]

(* The scheduler which backtracks in depth-first exploration *)
(* resume: resumption → stream data *)
and resume res =
 match res with
 [ [] → Void
 | [ React d q :: rest ] → react d q rest
 | [ Choose d ch del q' :: rest ] →
   match del () with (* we unfreeze the delayed stream of solutions *)
   [ Void → choose d ch rest (* finally we look for next pending choice *)
   | Stream d' del' → react d' q' [ Choose d ch del' q' :: rest ]
   ]
 ]
;

(* Note that these are just loops, since the recursive calls are terminal *)
```

```
(* Simulating the characteristic relation: relation data *)
value simulation d =
 let rec init_res l acc =
  match l with
  [ [] → acc
  | [ q :: rest ] → init_res rest [ React d q :: acc ]
  ] in
 resume (init_res initial [])
;

end; (* module Engine *)
```

5.2. Correctness, completeness, certification

Benoît Razet showed in [17] a formal proof of correctness and completeness of the simulation of a finite machine by the above reactive engine. Furthermore, it is possible to extract mechanically from this proof ML algorithms identical to the ones we showed above.

5.3. A General reactive engine, driven by a strategy

When a machine is not finite, and in particular when there are infinite computation paths, the bottom-up engine above may loop, and the simulation is not complete. In order to remedy this, we shall change the fixed last-in first-out policy of resumption management, and replace it by a more general strategy given as a parameter of the machine.

```
open Eilenberg;

module Engine (Machine: EMK) = struct
open Machine;

type choice = list (generator × state);

(* We separate the control choices and the data relation choices *)
type backtrack =
 [ React of data and state
 | Choose of data and choice
 | Relate of stream data and state
 ]
;
```

Now `resumption` is an abstract data type, given in a module `Resumption`, passed as argument to the `Strategy` functor, generalizing a backtrack stack.

```
module Strategy (* resumption management *)
 (Resumption : sig
   type resumption;
   value empty: resumption;
   value pop: resumption → option (backtrack × resumption);
   value push: backtrack → resumption → resumption;
 end) =
 struct

open Resumption;
```

Now we define a more parametric reactive engine, using an exploration strategy as parameter.

```
(* react: data → state → resumption → stream data *)
value rec react d q res =
 let ch = transition q in
 if accept q (* Solution d found? *)
   then Stream d (fun () → resume (push (Choose d ch) res))
   else resume (push (Choose d ch) res)

(* choose: data → choice → resumption → stream data *)
and choose d ch res =
 match ch with
 [ [] → resume res
 | [ (g, q') :: rest ] →
   let res' = push (Choose d rest) res in
   relate (semantics g d) q' res'
 ]

(* relate: stream data → state → resumption → stream data *)
and relate str q res =
 match str with
 [ Void → resume res
 | Stream d del → let str = del () in
                  resume (push (React d q) (push (Relate str q) res))
 ]

(* resume: resumption → stream data *)
and resume res =
 match pop res with
 [ None → Void
 | Some (b, rest) →
   match b with
   [ React d q → react d q rest
   | Choose d ch → choose d ch rest
   | Relate str q → relate str q rest
   ]
 ]
;

(* characteristic_relation: relation data *)
value simulation d =
 let rec init_res l acc =
   match l with
   [ [] → acc
   | [ q :: rest ] → init_res rest (push (React d q) acc)
   ] in
 resume (init_res initial empty)
;

end; (* module Strategy *)
```

5.4. A few typical strategies

We now give a few variations on search strategies. First of all, we show how the original depth-first reactive engine may be obtained by a `DepthFirst` strategy module, adequate for Finite Eilenberg Machines.

```
module DepthFirst = struct
 type resumption = list backtrack;
 value empty = [];
 value push b res = [ b :: res ];
 value pop res =
   match res with
   [ [] → None
   | [ b :: rest ] → Some (b,rest)
   ];
end; (* module DepthFirst *)
```

Next we examine the special case of sequential machines, where computations are deterministic. The following simple `Seq` tactic is adapted to this case.

```
module Seq = struct
 type resumption = list backtrack;
 value empty = [];
 value push b res =
   match b with
   [ React _ _ → [ b :: res ]
   | Choose _ _ → [ b ] (* cut : the list contains only one element *)
   | Relate _ _ → res (* no other delay *)
   ];
 value pop res =
   match res with
   [ [] → None
   | [ b :: rest ] → Some (b,rest)
   ];
end; (* module Seq *)
```

Finally, we show how to simulate in a fair way a general machine with a `Complete` tactic, which scans the state space in a top-down boustrophedon manner.

```
module Complete = struct
 type resumption = (list backtrack × list backtrack);
 value empty = ([],[]);
 value push b res =
   let (left,right) = res in
   (left, [ b :: right ])
 ;
 value pop res =
   let (left,right) = res in
   match left with
   [ [] → match right with
          [ [] → None
          | [ r :: rrest ] → Some (r, (rrest,[]))
          ]
   | [ l :: lrest ] → Some (l, (lrest,right))
```

```
  ]
 ;
end; (* module Complete *)
```

Now we may build the various modules encapsulating the various strategies.

```
module FEM = Strategy DepthFirst; (* The bottom-up engine *)
module Sequential_Engine = Strategy Seq; (* The sequential engine *)
module Complete_Engine = Strategy Complete; (* The fair engine *)

end; (* module Engine *)
```

6. From regular expressions to automata

Our motivation here is the design of a language for describing the control part of Eilenberg machines. The control part of Eilenberg machines is a finite automaton. It leads us naturally to *regular expressions* and their translations into finite automata.

There have been more than 50 years of research on the problem of compilation (or translation) of regular expressions into automata. It started with Kleene who stated the equivalence between the class of languages recognized by finite automata and the class of languages defined by regular expressions. This topic is particularly fruitful because it has applications to string search algorithms, circuits, synchronous languages, computational linguistics, *etc.* This wide range of applications leads to several automata and regular expressions variants.

Usually, an algorithm compiling regular expressions into automata is described in an imperative programming style for managing states and edges: states are allocated, merged or removed and so on concerning the edges. Surprisingly it seems that there is an applicative manner for describing each of the well-known algorithms. This methodology leads to a formal definition of the algorithm exhibiting important invariants. Of course we are careful to maintain the theoretical complexity of the algorithms.

We focus on fast translations, whose time complexity is linear or quadratic with respect to the size of the regular expression. First we present *Thompson*'s algorithm [18] and then we review other algorithms that are concerned by our methodology.

Let us mention Brzozowski's algorithm [5] which translates a regular expression (even with boolean operators) into a *deterministic* automaton. Unfortunately, the complexity is theoretically exponential. Nevertheless, it introduced the notion of regular expression *derivative* which is a fundamental idea pervading other algorithms.

6.1. Thompson's algorithm

Thompson presented his algorithm in 1968 and it is one of the most famous translations. It computes a finite non-deterministic automaton with ϵ-moves in linear time.

Let us first define regular expressions as the following datatype:

```
type regexp 'a =
  [ One
  | Symb of 'a
  | Union of regexp 'a and regexp 'a
  | Conc of regexp 'a and regexp 'a
  | Star of regexp 'a
  ];
```

The constructor `One` of arity 0 is for the 1 element of the corresponding action algebra. The following constructor `Symb` of arity 1 is the node for a generator. The type for the generator is abstract as expressed by the type parameter `'a` in the definition. The two following constructors are `Union` and `Conc` of arity 2 and describe union and concatenation operations. The last constructor `Star` is for the iteration or Kleene's star operator.

Now we have given the datatype for the input of our algorithm, let us present the datatype for the output (automata). We choose to implement states of the automaton with integers:

```
type state = int;
```

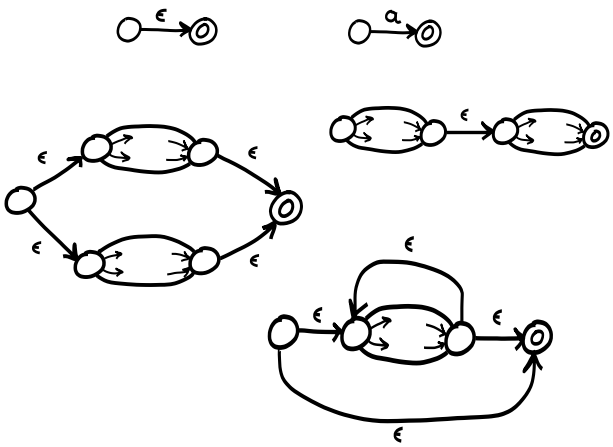
Automata obtained by Thompson's algorithm are non-deterministic and furthermore with ϵ-moves. We shall implement the control graph of such non-deterministic automata as a list of `fanout` pairs associating a list of labeled transitions to a state. This amounts to encoding a set of edges $s \xrightarrow{a} s'$ or triples (s, a, s') as an association list.

```
type fanout 'a = (state * list (label 'a * state))
and label 'a = option 'a
and transitions 'a = list (fanout 'a)
;
type automaton 'a = (state * transitions 'a * state);
```

A label is of type `option 'a` because it may be either an ϵ-move of value `None` or a generator `a` of value `Some a`. Note that even if they are non-deterministic, the automata we consider have only one initial and one accepting state.

We shall instanciate the `transition` function of the control component of our machines by composing the `transitions` list component of the constructed `automaton` with the primitive `List.assoc`, as we shall show later in section 7.

Thompson's algorithm can be summarized very shortly in a graphical way:



The algorithm performs a recursive traversal of the expression and each case corresponds to a drawing. It is presented in the order of the datatype definition: 1, generator, union, concatenation and Kleene's star.

```
(* thompson: regexp 'a → automaton 'a *)
value thompson e =
  let rec aux e t n =
    (* e is current regexp, t accumulates the state space,
       n is last created location *)
```

```
    match e with
    [ One → let n1=n+1 and n2=n+2 in
            (n1, [ (n1, [ (None, n2) ]) :: t ], n2)
    | Symb s → let n1=n+1 and n2=n+2 in
              (n1, [ (n1, [ (Some s, n2) ]) :: t ], n2)
    | Union e1 e2 →
      let (i1,t1,f1) = aux e1 t n in
      let (i2,t2,f2) = aux e2 t1 f1 in
      let n1=f2+1 and n2=f2+2 in
      (n1, [ (n1, [ (None, i1); (None, i2) ]) ::
                 [ (f1, [ (None, n2) ]) ::
                 [ (f2, [ (None, n2) ]) :: t2 ] ] ], n2)
    | Conc e1 e2 →
      let (i1,t1,f1) = aux e1 t n in
      let (i2,t2,f2) = aux e2 t1 f1 in
      (i1, [ (f1, [ (None, i2) ]) :: t2 ], f2)
    | Star e1 →
      let (i1,t1,f1) = aux e1 t n in
      let n1=f1+1 and n2=f1+2 in
      let t1' = [ (f1, [ (None, i1); (None, n2) ]) :: t1 ] in
      (n1, [ (n1, [ (None, i1); (None, n2) ]) :: t1' ], n2)
    ] in
  aux e [] 0
;
```

The algorithm constructs the automaton from the regular expression with a single recursive traversal of the expression. States are created at each node encountered in the expression: each constructor creates 2 states except the concatenation `Conc` that does not create any state. Remark the invariant of the recursion: each regular subexpression builds an automaton (i, fan, f) with $0 < i < f$ and $dom(fan) = [k..f-1]$. States are allocated so that disjoint subexpressions construct disjoint segments $[i..f]$. This invariant of the `thompson` function implies that we have to add finally a last (empty) fanout for the final state.

```
(* thompson_alg: regexp 'a → automaton 'a *)
value thompson_alg e =
  let (i,t,f) = thompson e in
  (i, [(f,[]) :: t], f)
;
```

The function `thompson_alg` implements Thompson's algorithm in linear time and space because it performs a unique traversal of the expression.

6.2. Other algorithms

We have seen that Thompson's algorithm is linear and can be implemented in an applicative manner. Let us mention also Berry-Sethi's algorithm [3] that computes a non-deterministic automaton (without ϵ-move), more precisely a *Glushkov* automaton. This construction is quadratic and we provided an implementation of it in ML [12]. In 2003, Ilie and Yu [13] introduced the Follow automata which are also non-deterministic automata. Champarnaud, Nicart and Ziadi showed in 2004 [6] that the Follow automaton is a quotient of the one produced by the Berry-Sethi algorithm (i.e. some states are merged together). They also provide an algorithm implementing the Follow construction in quadratic time. The applicative implementation of the Berry-Sethi algorithm may be extended to yield the Follow

automaton. Also, in 1996 Antimirov proposed an algorithm [2] that compiles even smaller automata than the ones obtained by the Follow construction, provided the input regular expression is presented in *star normal form* (as described in [4]). The algorithm presented originally was polynomial in $O(n^5)$ but Champarnaud and Ziadi [7] proposed yet another implementation in quadratic time.

It is possible to validate these various compiling algorithms using some of the algebraic laws of action algebras we presented in Section2. In particular, use of idempotency to collapse states will indicate that the corresponding construction does not preserve the notion of multiplicity of solutions. Furthermore, such a notion of multiplicity, as well as weighted automata modeling statistical properties, generalise to the treatment of valuation semi-rings, for which Allauzen and Mohri [1] propose extensions of the various algorithms.

7. Working out an example

We briefly discussed above how to implement as a machine a finite automaton recognizing a regular language. We may use for instance Thompson's algorithm to compile the automaton from a regular expression defining the language. This example will show that recognizing the language and generating the language are two instances of machines which share the same control component, and vary only on the data domain and its associated semantics. Furthermore, we show in the recognition part that we may compute the multiplicities of the analysed string. However, note that this is possible because Thompson's construction preserves this notion of multiplicity.

Let us work out completely this method with the regular language defined by the regular expression $(a^*b + aa(b^*))^*$.

```
(* An example: recognition and generation of a regular language L *)

(* L = (a*b |aa(b)* )* *)
value exp =
  let a = Symb 'a' in
  let b = Symb 'b' in
  let astarb = Conc (Star a) b in
  let aabstar = Conc a (Conc a (Star b)) in
  Star (Union astarb aabstar)
;
value (i,fan,t) = thompson_alg exp
;
value graph n = List.assoc n fan
;
value delay_eos = fun () → Void
;
value unit_stream x = Stream x delay_eos
;

module AutoRecog = struct
 type data = list char;
 type state = int;
 type generator = option char;
 value transition = graph;
 value initial = [ i ];
 value accept s = (s = t);
 value semantics c tape = match c with
```

```
  [ None → unit_stream tape
  | Some c → match tape with
    [ [] → Void
    | [ c' :: rest ] → if c = c' then unit_stream rest else Void
    ]
  ];
end (* AutoRecog *)
;
module LanguageDeriv = Engine AutoRecog
;
(* The Recog module controls the output of the sub-machine
   LanguageDeriv, insuring that its input is exhausted *)
module Recog = struct
 type data = list char;
 type state = [ S1 |S2 |S3 ];
 type generator = int;
 value transition = fun
   [ S1 → [ (1,S2) ]
   | S2 → [ (2,S3) ]
   | S3 → []
   ];
 value initial = [ S1 ];
 value accept s = (s = S3);
 value semantics g tape = match g with
   [ 1 → LanguageDeriv.Complete_Engine.simulation tape
   | 2 → if tape = [] then unit_stream tape else Void
   | _ → assert False
   ];
end (* Recog *)
;
module WordRecog = Engine Recog
;
module AutoGen = struct
 type data = list char;
 type state = int;
 type generator = option char;
 value transition = graph;
 value initial = [ i ];
 value accept s = (s = t);
 value semantics c tape =
   match c with
   [ None → unit_stream tape
   | Some c → unit_stream [ c :: tape ]
   ];
end (* AutoGen *)
;
module AutoGenBound = struct
 type data = (list char * int); (* string with credit bound *)
 type state = int;
 type generator = option char;
 value transition = graph;
```

```
 value initial = [ i ];
 value accept s = (s = t);
 value semantics c (tape, n) =
   if n < 0 then Void
   else match c with
     [ None → unit_stream (tape, n)
     | Some c → unit_stream ([ c :: tape ], n-1)
     ];
end (* AutoGenBound *)
;
module WordGen = Engine AutoGen;
module WordGenBound = Engine AutoGenBound;

(* Service functions on character streams for testing *)

(* print char list *)
value print_cl l =
 let rec aux l = match l with
   [ [] → ()
   | [ c :: rest ] → let () = print_char c in aux rest
   ] in
 do { aux l; print_string "\n" }
;
value iter_stream f str =
 let rec aux str = match str with
   [ Void → ()
   | Stream v del → let () = f v in aux (del ())
   ] in
 aux str
;
value print_cl2 (tape,_) = print_cl tape
;
value cut str n =
 let rec aux i str =
   if i ≥ n then Void
   else match str with
     [ Void → Void
     | Stream v del → Stream v (fun () → aux (i+1) (del ()))
     ] in
 aux 0 str
;
value count s =
  let rec aux s n =
    match s with
    [ Void → n
    | Stream _ del → aux (del ()) (n+1)
    ] in
 aux s 0
;
(* Now we show typical invocations: *)
print_string "Recognition␣of␣word␣‘aaaa’␣with␣multiplicity:␣";
```

```
print_int (count (WordRecog.FEM.simulation ['a' ; 'a' ; 'a' ; 'a' ]));
print_newline ();
print_string "Recognition␣of␣word␣`aab'␣with␣multiplicity:␣";
print_int (count (WordRecog.FEM.simulation ['a' ; 'a' ; 'b' ]));
print_newline ();
(* Remark that we generate mirror images of words in L *)
print_string "First␣10␣words␣in␣~L␣in␣a␣complete␣enumeration:\n";
iter_stream print_cl (cut (WordGen.Complete_Engine.simulation []) 10);
print_string "All␣words␣in␣~L␣of␣length␣bounded␣by␣3:\n";
iter_stream print_cl2 (WordGenBound.FEM.simulation ([],3));
```

The output of executing the above code is shown below:

```
Recognition of word `aaaa' with multiplicity: 1
Recognition of word `aab' with multiplicity: 3
First 10 words in ~L in a complete enumeration:
b
ba
aa
baa
baa
baaa
bbaa
bb
baaaa
All words in ~L of length bounded by 3:
baa
bba
ba
bab
bbb
bb
aab
b
baa
baa
aa
```

Conclusion

We presented a general model of non-deterministic computation based on a computable version of Eilenberg machines. Such relational machines complement a non-deterministic finite state automaton over an alphabet of relation generators with a semantics function interpreting each relation functionally as a map from data elements to streams of data elements. The relations thus computed form an action algebra in the sense of Pratt. We survey some algorithms which permit to compile the control component of our machines from regular expressions. The data component is implemented as an ML module consistent with an `EMK` interface. We show how to simulate our non-deterministic machines with a reactive engine, parameterized by a strategy. Under appropriate fairness assumptions of the strategy the simulation is complete. An important special case is that of finite machines, for which the bottom-up strategy is complete, while being efficiently implemented as a flowchart algorithm.

Bibliographie

[1] C. Allauzen and M. Mohri. A unified construction of the Glushkov, Follow, and Antimirov automata. *Springer-Verlag LNCS*, 4162:110–121, 2006.

[2] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[3] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.

[4] A. Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.

[5] J. A. Brzozowski. Derivatives of regular expressions. *J. Assoc. Comp. Mach.*, 11(4):481–494, October 1964.

[6] J.-M. Champarnaud, F. Nicart, and D. Ziadi. Computing the follow automaton of an expression. In *CIAA*, pages 90–101, 2004.

[7] J.-M. Champarnaud and D. Ziadi. Computing the equation automaton of a regular expression in $o(s^2)$ space and time. In *CPM*, pages 157–168, 2001.

[8] S. Eilenberg. *Automata, Languages, and Machines, volume A*. Academic Press, 1974.

[9] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27,4:797–821, 1980.

[10] G. Huet. The Zen computational linguistics toolkit: Lexicon structures and morphology computations using a modular functional programming language. In *Tutorial, Language Engineering Conference LEC'2002*, 2002.

[11] G. Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional Programming*, 15,4:573–614, 2005.

[12] G. Huet and B. Razet. The reactive engine for modular transducers. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 355–374. Springer-Verlag LNCS vol. 4060, 2006.

[13] L. Ilie and S. Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, 2003.

[14] D. Kozen. On action algebras. In J. van Eijck and A. Visser, editors, *Logic and Information Flow*, pages 78–88. MIT Press, 1994.

[15] V. Pratt. Action logic and pure induction. In *Workshop on Logics in Artificial Intelligence*. Springer-Verlag LNCS vol. 478, 1991.

[16] B. Razet. Finite Eilenberg machines. In O. Ibarra and B. Ravikumar, editors, *Proceedings of CIIA 2008*, pages 242–251. Springer-Verlag LNCS vol. 5148, 2008. `http://gallium.inria.fr/~razet/fem.pdf`

[17] B. Razet. Simulating finite Eilenberg machines with a reactive engine. In *Proceedings of MSFP 2008*. Electric Notes in Theoretical Computer Science, 2008. `http://gallium.inria.fr/~razet/PDF/razet_msfp08.pdf`

[18] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

Présentation de SSReflect

Assia Mahboubi

INRIA Saclay - Île-de-France

L'équipe Composants Mathématiques du centre commun INRIA Microsoft Research travaille au développement d'un grand corpus de bibliothèques formelles modulaires pour le système Coq. Le but de cette expérience est de comprendre comment mener à bien le développement de bibliothèques de mathématiques formalisées, réutilisables et combinables. Il s'agit de transposer les méthodes de génie logiciel modernes à l'organisation de théories formalisées en composants. Par composants, on entend ici des modules qui incluent à la fois le contenu statique (objets et propriétés) et le contenu dynamique (preuves et méthodes de calcul) des théories.

Ce projet s'appuie sur la preuve formelle du théorèmes de quatre couleurs dans l'assistant à la preuve Coq, qui a été achevée en 2004 par G. Gonthier, avec l'aide de B. Werner. Le but de ce projet est cette fois de construire une preuve formelle du théorème de Feit-Thompson (1963). Ce pan de la classification des groupes finis constitue un passage a l'échelle significatif pour les contributions issues de la preuve du théorèmes des quatre couleurs. En particulier, cette preuve exige une combinaison de théories formalisées jusqu'ici peu voire jamais combinées.

Dans ce cours, on présentera les techniques de développement de preuve formelle qui se dégagent de ces expériences : choix des structures de données, réflexion à petite échelle, exploitation de l'inférence de types, extension du langage de tactique du système Coq . . . ainsi qu'un panorama des bibliothèques actuellement développées.

Ocsigen : approche fonctionnelle typée de la programmation Web

Vincent Balat

Université Paris 7

Vers la début des années 2000, Christian Queinnec, John Hughes et Paul Graham ont, indépendamment, montré une application intéressante du concept de *continuation* utilisé en programmation fonctionnelle, pour décrire le fonctionnement du bouton « back » dans une application Web. L'utilisation de cette découverte permet de simplifier l'implémentation de certains comportements et d'éviter des erreurs courantes.

En fait ce premier effort d'abstraction peut être poussé beaucoup plus loin. Il est en effet possible d'inventer des concepts de haut niveau pour modéliser d'autres comportements courants sur le Web, liés par exemple aux sessions ou à d'autres modes d'interaction avec l'utilisateur par l'intermédiaire d'une interface Web.

Nous proposons d'abord de définir un nouveau langage vernaculaire pour décrire l'interaction Web, en s'appuyant sur la sémantique plutôt que sur les contraintes technologiques.

L'implémentation directe de ces concepts abstraits dans des outils de programmation Web apporte un fort gain d'expressivité, en simplifiant la programmation de comportements complexes. Cela peut aboutir notamment à une meilleur ergonomie. Cette abstraction permet également de fortes garanties statiques qui améliorent la robustesse des applications.

Il en découle un nouveau style de programmation Web qui répond parfaitement aux besoins liés à l'évolution actuelle du Web. Nous en montrerons une implémentation en OCaml sous la forme d'un module pour le serveur Web Ocsigen, appelé Eliom.

Qui sème la fonction, récolte le tuyau typé

Didier Parigot & Bernard Paul Serpette

Inria Sophia-Antipolis - Méditerranée
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
First.Last@inria.fr
http://www-sop.inria.fr/members/First.Last

Résumé

Les applications de l'internet de demain vont devoir communiquer avec des objets de plus en plus complexes. Actuellement, cette communication s'effectue essentiellement à l'aide de protocoles de communication qui sont par nature des mécanismes mal typés. De plus, à chaque avancée technologique (sans fils, mobile...), de nouveaux types de protocoles de transport apparaissent.

L'objectif de l'article est de proposer un mécanisme de création de tuyau pour une communication bien typée et pour abstraire le protocole sous-jacent à la communication. L'idée de base, afin d'établir un tuyau, est de migrer du serveur vers le client une fonction qui se charge de mettre en place la structure du tuyau. Cette fonction permet au client de s'abstraire du protocole de communication. Ce tuyau, une fois établi, pourra aussi faire véhiculer des fonctions, cette fois-ci du client vers le serveur. Ces fonctions permettent au client d'exprimer l'échange d'information en termes d'expressions du langage source et donc d'assurer le typage de la communication. D'une autre manière, ces fonctions donnent les moyens au client de communiquer directement avec le serveur en cachant les détails du protocole puisque ces fonctions seront exécutées finalement sur le serveur.

Avec cette notion de tuyau, on définit un protocole comme étant un générateur de tuyau bien typé. Nous montrerons qu'il est possible d'établir des tuyaux de communication bien typés vers tous les objets atteignables sur un réseau.

1. Introduction

Le projet LogNet de l'Inria s'intéresse aux réseaux logiques : des structures distribuées où un programme peut s'intégrer, demander des ressources et activer ces ressources. Les entités participant au réseau communiquent entre elles pour assurer la fonctionnalité du réseau. Dans ce cadre, nous avons commencé le développement d'une bibliothèque Java, nommée Pinet, définissant l'interface d'un réseau en terme d'inscription et de requête, et des implémentations de cette interface. Une des implémentations visées est le système Arigatoni [1], mais nous visons aussi d'autres structures de réseaux comme celle de Chord [9]. Le réseau, quelque soit son implémentation, est utilisé, de manière abstraite, par le système orienté services de SmarTools [4] pour connecter les agents entre eux.

Le noyau de Pinet propose une structure abstraite de *tuyau* pour définir les communications entre les agents du réseau. Un tuyau est l'association d'un lecteur et d'un écrivain : les deux bouts du tuyau. L'implémentation du tuyau définit une structure qui véhiculera les données écrites par l'écrivain, vers le lecteur qui les lira. Il y a beaucoup de structures de tuyau, un fil reliant deux portes du microprocesseur est un tuyau véhiculant des bits, un registre, une variable C ou Scheme, une référence Caml, une queue, une pile, un couple de descripteurs de ficher, un flux TCP ; toutes ces structures sont des tuyaux. Nos travaux portent essentiellement sur les flux connectant deux programmes sur deux machines différentes.

Dans cet article nous ne nous attarderons pas sur la structure des tuyaux, mais nous analyserons les lecteurs et les écrivains d'un point de vue du typage. Un lecteur est une fonction de type $unit \rightarrow \alpha$ et un écrivain une fonction de type $\beta \rightarrow unit$. Le rôle du tuyau, via sa structure, est d'unifier l'α et le β. Il va garantir, statiquement, que les valeurs écrites sont du même type que celles qui sont lues. Lorsque le tuyau est connecté sur une même mémoire, cette contrainte est facilement assurée par le vérificateur de type. Dans le cadre du calcul réparti, cette vérification est moins immédiate : le lecteur et l'écrivain sont souvent écrits dans des programmes indépendants (module serveur et module client par exemple). Pour résoudre ce problème, il suffit d'instancier le lecteur et l'écrivain dans un même programme et de transférer l'écrivain sur la machine devant faire les écritures. Malheureusement, les écrivains ne sont pas transférables d'une machine à l'autre, ils manipulent des informations liées à la structure du tuyau : descripteurs de fichiers, *sockets*...Cette fois ci, le problème est résolu en transférant un générateur d'écrivains : une fonction qui créera l'écrivain sur la machine cible, i.e. celle qui ouvrira la *socket*. Ceci est la première raison qui nous a amené à considérer la migration de fonctions : dans un même programme sont créés un lecteur et une fonction pouvant générer un écrivain, cette fonction migrera sur le réseau pour instancier l'écrivain au bon endroit.

D'un autre coté, on peut remarquer que derrière les protocoles de transport sur un réseau physique se cache généralement un interprète, simple pour HTTP avec les fonctions GET et POST ; plus complexe pour le protocole Arigatoni. Quoi qu'il en soit, le principe reste que le client écrit une expression E qui sera décodée, i.e. interprétée, par le serveur. L'idée est de donner aux expressions tout le potentiel de celles d'un langage de haut niveau. L'expression E peut donc apparaître textuellement dans le code du client. Pour éviter que E soit exécutée sur le client, il faut utiliser une technique d'évaluation retardée, ou glaçon, qui consiste principalement à transformer E en une fonction $\lambda().E$. Rajouter un paramètre formel à la fonction permet d'abstraire le serveur du coté client, si E doit s'exécuter sur un serveur s dont on désire certaines fonctionnalités, i.e. s est libre dans E, alors $\lambda s.E$ est une fonction que l'on peut essayer de typer dans un programme client et qui peut-être écrite dans un tuyau vers le serveur s pour être auto-appliquée. Ceci nous a encore amené à considérer la migration des fonctions.

Cet article s'organise comme suit : dans un premier temps nous allons montrer, par un exemple Java, comment transmettre une fonction permettant d'abstraire l'objet serveur ; dans une deuxième partie, nous analyserons le type Java des tuyaux pour leur donner une spécification fonctionnelle ; dans une troisième partie nous définirons, en Caml, des générateurs de tuyaux pour construire une classe d'équivalence basée sur la relation "*est connecté à*" ; puis nous reviendrons sur la création des tuyaux initiaux avant de conclure.

2. Le principe

Montrons, par un exemple simple, une communication typée et comment elle se différencie de RMI (*Java Remote Method Invocation*)[10] qui est la version orientée objet de RPC (*Remote Procedure Call*)[3]. L'exemple consiste à instancier un objet proposant un service, ici une simple méthode `hello` affichant un message à l'écran, et de montrer comment on active ce service à partir d'une autre machine. L'objet qui expose le service est communement appelé *le serveur* et l'objet qui requiert ce service *le client*.

En RMI, les services à exposer doivent passer par une interface.

```
public interface RmiI extends Remote {
    public void hello() throws RemoteException;
}
```

C'est cette interface qui sera visible par le client RMI. Le type `Remote`, super-classe de l'interface, cache la machinerie RMI qui fait que des objets seront accessibles de l'extérieur. Il convient maintenant

de définir une implémentation de cette interface.

```
public class RmiS implements RmiI {
    public void hello() {
        System.out.println("Hello");
    }
}
```

Le même code sera pris pour Pinet, hormis le fait qu'il n'est pas nécessaire de passer par une interface. Il est indéniable que cette interface permet de cacher l'implémentation du service au client, mais il est des cas où cette implémentation n'a pas à être cachée, ce sont plus les données manipulées par le serveur qui font sa particularité et qui empêche le service d'être exécuté sur le client : l'accessibilité d'une base de données par exemple. Pinet peut aussi passer par une interface, par contre RMI ne peut pas passer par une classe.

```
public class PinetS {
    public void hello() {
        System.out.println("Hello");
    }
}
```

Le plus important est que, dans cette définition de la classe `PinetS`, ou dans celles des interfaces qu'elle pourrait implémenter, rien ne montre que les instances de cette classe pourront ou devront être accessibles de l'extérieur. Toute classe Java peut potentiellement exposer ses méthodes à l'extérieur.

Le serveur doit maintenant créer une intance et l'exposer à l'extérieur :

```
public static void main(String args[]) {
    publishRMI(args[0], new RmiS());              /* Publication RMI   */
    publishPinet(PinetS.class, new PinetS());     /* Publication Pinet */
}
```

Dans les deux cas, RMI et Pinet, on crée l'objet à exposer et on utilise un système de publication. Ici nous voyons la première différence qui fait que Pinet expose les objets sous une clé qui représente précisément le type de l'objet exposé, alors que RMI utilise une chaîne de caractères [1] ce qui a posteriori, du coté client, va nécessiter une vérification dynamique du type de l'objet exposé.

En Java, `T.class` représente l'objet "*classe de* `T`" et est de type `Class<T>`. Ainsi la signature de la fonction `publishPinet` est : `<T> void publishPinet(Class<T>, T)`. Selon la notation décrite dans Henry & al[7], `T.class` s'écrirait en Caml : `<<T>>` et la signature de `publishPinet` deviendrait : $\forall\alpha.\texttt{TRepr}(\alpha) \rightarrow \alpha \rightarrow unit$.

Du coté client, il faut, à partir de la clé, rechercher un objet susceptible d'activer le service et activer ce service de manière appropriée :

```
public static void main(String[] args) {
    ((RmiI) lookupRMI(args[0])).hello();      /* Recherche et activation RMI  */
    lookupPinet(PinetS.class).write(new Proc1<PinetS>() { /* Recherche Pinet  */
            public void call(PinetS s) {
                s.hello();                                /* Activation Pinet */
            }});
}
```

1. L'expression *(publishRMI name obj)* est équivalente à *LocateRegistry.getRegistry().rebind(name, UnicastRemoteObject.exportObject(obj, 0))*

La seconde différence, entre RMI et Pinet, concerne le résultat de la recherche qui est, pour RMI[2], un objet fournissant virtuellement les services requis et, pour Pinet, un objet permettant de communiquer avec l'objet fournissant réellement les services requis. Derrière l'objet de type `RmiI`, reçu par le client, se trouve toute la machinerie de transport sur le réseau : le code de la méthode `hello` de l'objet obtenu par le client envoie, via le réseau, un message qui activera la véritable méthode sur le serveur. Tout ce code spécifique est généré par compilation (`rmic`) ou automatiquement depuis la version 1.5. Ce code est appelé *talon* ou *stub* et se trouve pour une partie sur le client, pour l'autre sur le serveur.

Pinet propose une solution sans génération de code talon. Le client demande une valeur associée au *type* du serveur recherché, le résultat de cette requête est le bout de ce que nous appellerons un *tuyau* sur lequel il est possible d'écrire une procédure, i.e. une fonction ne retournant pas de résultat. Les procédures écrites dans le tuyau seront exécutées par la machine du serveur et recevront ce serveur en paramètre. Ainsi, l'expression `s.hello()` est écrite dans le programme client mais sera interprétée par le serveur.

Pour Pinet, la verbosité de code pour activer le service s'explique par le fait que Java n'a pas la notion de fonction anonyme. Le code `"new Proc1<PinetS>() {public void call XXX}"` devrait se réécrire, ou du moins doit se lire comme : `Lambda XXX`. Ainsi le code du client pourrait se réécrire, pour Scheme, en `(write (lookupPinet PinetS) (lambda (s) (hello s)))` ou, pour Caml, en `(write (lookupPinet <<PinetS>>) (fun (s) -> (hello s)))`.

Nous allons maintenant décrire la structure des tuyaux.

3. Le type des tuyaux

Dans l'exemple vu précédemment, nous n'avons qu'une partie des tuyaux : le résultat d'une recherche sur le réseau est le bout d'un tuyau sur lequel on peut écrire. L'autre bout du tuyau a été construit par le serveur au moment de la publication du service. Ainsi, un tuyau contient deux bouts, d'un coté on écrit/envoie, de l'autre on lit/reçoit. La seule contrainte que l'on voudrait imposer est que, pour un même tuyau, les valeurs écrites et lues soient de même type. C'est une contrainte assez forte que l'on retrouve dans les langages typés sous le fait que l'on ne peut pas avoir de structures répétitives (liste, vecteur, ensemble ...) hétérogènes, i.e. une liste contenant indifféremment des entiers et des fonctions. Mais, derrière cette restriction, on a l'assurance que le programme ne peut pas échouer du fait qu'une valeur d'un mauvais type ait été écrite dans un tuyau.

En Java, les lecteurs et les écrivains sur les tuyaux sont décrits par des classes abstraites paramétrées par le type des valeurs circulant dans le tuyau :

```
public abstract class InFlow<T> {
    public abstract T read();
}

public abstract class OutFlow<T> {
    public abstract void write(T value);
}
```

Comme ces deux classes ne proposent qu'une seule méthode et n'ont pas de données, elles sont équivalentes à de simples fonctions. Un lecteur est donc une fonction de type $unit \rightarrow \alpha$ et un écrivain une fonction de type $\alpha \rightarrow unit$.

Les lecteurs et écrivains sont certainement des objets encapsulant des données spécifiques au transport : la *socket* résultant d'une connexion TCP par exemple. Ces données n'ont de sens que pour

2. *lookupRMI* est un alias de *java.rmi.Naming.lookup*

la machine qui les possède, elles ne sont pas transmissibles, on dit alors qu'elles sont *non sérialisables*. Par contre, ces lecteurs et écrivains sont construits à partir de valeurs plus simples (nom de machine, numéro de port ...) qui sont sérialisables. Il est donc possible d'encapsuler ces valeurs simples dans des objets qui vont circuler sur le réseau et pouvoir générer les écrivains à l'endroit souhaité. En Java, un générateur d'écrivains est encore décrit par une classe abstraite paramétrée par le type des valeurs circulant dans le tuyau :

```
public abstract class OutFlowGenerator<T> implements Serializable {
    public abstract OutFlow<T> generate();
}
```

Ici, on mentionne explicitement, par le mot clé `Serializable`, le fait que les générateurs d'écrivains peuvent circuler sur le réseau. De la même manière que pour les lecteurs et les écrivains, les générateurs d'écrivains sont équivalents à une simple fonction de type $unit \rightarrow OutFlow(\alpha)$, soit : $unit \rightarrow \alpha \rightarrow unit$.

L'étape suivante consiste à associer un lecteur avec un générateur d'écrivains de même type, ce qui donne en Java :

```
public class EndPointFlow<T> {
    public InFlow<T> inFlow;
    public OutFlowGenerator<T> outGenerator;
}
```

Ici la classe ne définit que deux champs sans implémenter de méthodes. Cette structure est un produit cartésien : $Inflow(\alpha) * OutFlowGenerator(\alpha)$, soit le type $(unit \rightarrow \alpha) * (unit \rightarrow \alpha \rightarrow unit)$. C'est ce type qui permet d'unifier le type paramétrique du lecteur avec celui de l'écrivain, i.e. le même α. Ce produit cartésien, contenant directement un lecteur, n'est pas sérialisable. Pour ce faire, il suffit de définit un générateur de produit cartésien :

```
public abstract class Protocol implements Serializable {
   public abstract <T> EndPointFlow<T> generate();
}
```

Avec le même principe que précédemment, cette classe est équivalente au type : $unit \rightarrow (unit \rightarrow \alpha) * (unit \rightarrow \alpha \rightarrow unit)$. Toute fonction ayant ce type sera considérée comme un protocole. Nous allons maintenant donner deux exemples de protocoles, l'un en OCaml, l'autre en Java.

3.1. Exemples de protocoles

Voici un protocole défini en Objective Caml :

```
let qprot () =
   let q = Queue.create () in
      (fun () -> (Queue.pop q)),
      (fun () -> (fun (v) -> (Queue.push v q)))
```

Ce protocole n'est pas très intéressant du fait qu'il ne fonctionne qu'en mémoire partagé et que la lecture n'est pas bloquante sur une queue vide, mais l'important est que la fonction a bien le type des protocoles. On voit ici le principe général des protocoles, qui, à leur activation, génère la structure du tuyau, ici une queue, qui sera utilisée par le lecteur, `(fun () -> (Queue.pop q))`, et par l'écrivain,

`(fun (v) -> (Queue.push v q))`. Par contre, la structure du tuyau, i.e. la queue, n'apparaît pas dans le type des protocoles.

Dans cette définition du protocole `qprot`, on ne voit pas la nécessité du générateur d'écrivains. Ici, l'écrivain `(fun (v) ...)` est sérialisable. Généralement, les écrivains ne sont pas sérialisables, voici une partie du code Java lié au protocole TCP :

```
public class Tcp extends Protocol {
    public <T> EndPointFlow<T> generate() {
        ServerSocket serversocket = new ServerSocket(0);
        return(new EndPointFlow<T>(
            new TcpSeqInFlow<T>(serversocket),
            new TcpOutFlowGenerator<T>(InetAddress.getLocalHost(),
                                       serversocket.getLocalPort())));
}   }

class TcpOutFlowGenerator<T> extends OutFlowGenerator<T> {
    private InetAddress where;
    private int port;
    public OutFlow<T> generate() {
        return(new TcpOutFlow<T>(new Socket(where, port).getOutputStream()));
}   }

class TcpOutFlow<T> extends OutFlow<T> {
    private ObjectOutputStream out;
    public void write(T value) {
        out.writeObject(value);
        out.flush();
}   }
```

On voit ici que le générateur d'écrivains `TcpOutFlowGenerator` ne contient que des valeurs sérialisables, à savoir l'adresse d'une machine et un port. L'écrivain effectif `TcpOutFlow` sera construit à partir de ces valeurs sérialisables. L'écrivain sera opérationnel là où le générateur aura été transmis et activé.

Pour en revenir au type des protocoles, il n'est pas indispensable que le lecteur et le générateur d'écrivains aient le même type polymorphe. Seuls les protocoles liés au transport physique sur le réseau doivent avoir cette particularité. Par exemple, si on cherche à cryptographier les valeurs transmises sur le réseau connaissant un protocole pour les valeurs cryptographiées, une fonction de cryptographie et son inverse, voici une fonction qui calcule le nouveau protocole :

```
let secure prot crypt uncrypt =
   let (read,writer)=(prot ()) in
      (fun () -> uncrypt (read ())),
      (fun () -> (fun v -> ((writer ()) (crypt v))))
```

Cette fonction `secure` crypte les valeurs avant de les donner à l'écrivain et décrypte les valeurs lues par le lecteur. Cette fonction est très polymorphe, son type est :

$(unit \rightarrow (unit \rightarrow \alpha) * (unit \rightarrow \beta \rightarrow \gamma)) \rightarrow (\delta \rightarrow \beta) \rightarrow (\alpha \rightarrow \epsilon) \rightarrow (unit \rightarrow \epsilon) * (unit \rightarrow \delta \rightarrow \gamma)$

Si on la spécialise avec un protocole de base comme `qprot` cela permet d'unifier α et β, donc le type de `secure qprote` est :

$(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (unit \rightarrow \gamma) * (unit \rightarrow \alpha \rightarrow unit)$

Et l'on obtient bien comme contrainte que la sortie de la fonction de cryptographie, β, doit correspondre à l'entrée de son inverse. Par contre, le lecteur et le générateur d'écrivains du résultat ne sont plus en relation : on lit des valeurs de type γ et on écrira des valeurs de type α.

Nous avons vu que les tuyaux pouvaient être vus comme des objets fonctionnels et qu'ils étaient typables. Nous allons voir maintenant comment, en utilisant des tuyaux faisant transiter des fonctions, on peut créer de nouveaux tuyaux bien typés.

4. Génération de tuyaux

Dans cette section, nous allons présenter un ensemble de générateurs de tuyaux. Toutes ces fonctionnalités prennent en argument (paramètre `out`) un écrivain de procédure sur un type `T`, par exemple le résultat de l'appel `lookupPinet(T.class)` que nous avons vu dans le premier exemple. Selon les types fonctionnels que nous avons vus précédemment, cet écrivain aura le type $(\alpha \rightarrow unit) \rightarrow unit$.

La première primitive indispensable est de pouvoir activer une *fonction* à distance plutôt qu'une *procédure*. Il faut donc faire revenir vers le client la valeur produite par la fonction. Si `out` est un écrivain de procédure ($(\alpha \rightarrow unit) \rightarrow unit$), si `f` est une fonction conforme à l'écrivain ($\alpha \rightarrow \beta$) et si `prot` est un protocole qui lit et écrit des valeurs de type β, alors la fonction suivante active `f` via `out` et retourne un lecteur du protocole `prot` avec lequel on peut lire la valeur calculée par `f`.

```
let detach prot out f  =
   let (read,writer)=(prot ()) in
      (out (fun (s) -> ((writer ()) (f s))));
      read
```

Dans un premier temps, on crée sur le client le lecteur (`read`) et le générateur d'écrivains (`writer`) pour la valeur de retour. Puis on écrit sur le tuyau `out` une procédure qui sera activée sur le serveur `s`. Cette procédure génère l'écrivain (`writer ()`), appelle la fonction `f` avec le serveur en argument (`(f s)`) et transmet le résultat de cet appel via l'écrivain. Ce résultat pourra être lu par le lecteur `read` qui est retourné par la fonction `detach`.

Comme la valeur de la variable `writer` est fermée dans la procédure qui est transmise via `out`, cette valeur transitera en même temps que la procédure. Ceci montre la nécessité que les générateurs d'écrivains soient sérialisables. De la même manière, la valeur fonctionnelle `f` doit aussi être sérialisable, tout autant que la valeur produite par `f`.

À titre d'indication, cette fonction `detach` nécessite trois fois plus de lignes pour être écrite en Java...

Dans la terminologie du calcul concurrent, la fonction `detach` engendre un appel asynchrone : elle n'attend pas le résultat de l'application effectuée à distance. La synchronisation se fait par activation du lecteur. Voici comment définir un appel synchrone.

```
let call prot out f = ((detach prot out f) ())
```

Cette fonction est très polymorphe, mais si on la spécialise avec un protocole et un écrivain (on prendra `write = ((snd (qprot ())) ())`), alors le type de `call qprot write` a exactement le type que l'on souhaite : $(\alpha \rightarrow \beta) \rightarrow \beta$.

On peut généraliser cette fonction par une fonction de symétrie qui, connaissant un serveur via un écrivain `out`, peut se faire connaître de ce serveur via un nouvel écrivain :

```
let sym prot out connect =
   let (read,writer)=(prot ()) in
      (out (fun (s) -> (connect s (writer ()))));
      read
```

Les deux premiers arguments de la fonction `sym` ont le même rôle que ceux de `detach`, et la notion "*se faire connaître via un nouvel écrivain*" est concrétisée par le dernier argument, `connect`, qui est une fonction recevant en paramètre le serveur et l'écrivain. Ainsi la fonction `detach` peut se redéfinir :

```
let detach prot out f = sym prot out (fun s w -> (w (f s)));
```

De la même manière, on peut définir une fonction de transitivité qui, connaissant un serveur `a` via un écrivain `out`, sachant que `a` connaît un autre serveur `b`, construit directement un tuyau vers `b` dont l'écrivain sera rendu en résultat.

```
let trans prot out get connect repp =
   let (read,writer)=(repp ()) in
      (out (fun (a) -> ((get a) (fun (b) -> let (readb,writerb)=(prot ()) in
                                              (connect b readb);
                                              ((writer()) writerb) ))));
      ((read ()) ())
```

Le premier argument, `prot`, est le protocole du tuyau que l'on veut construire, il transitera, via les fermetures, sur le serveur `a` puis sur le serveur `b`. Le second, `out`, est un écrivain de procédure sur le serveur `a`, le troisième, `get`, est une fonction qui, à partir du serveur `a`, trouve un écrivain de procédure sur le serveur `b`. Le quatrième, `connect`, est une fonction activée sur le serveur `b` qui prévient ce dernier de l'existence du lecteur sur le tuyau que l'on a créé. Le dernier, `repp`, est un protocole pour créer un tuyau temporaire.

La fonction `trans`, crée un lecteur et un générateur d'écrivains temporaire. Ce générateur va transiter, via les fermetures, sur le serveur `b`. Puis elle écrit, via le tuyau `out`, une procédure qui sera activée sur `a`. Cette procédure recherche, via la fonction `get`, un tuyau sur `b` et écrit sur ce dernier une nouvelle proédure. Celle-ci, une fois sur `b`, active le protocole pour obtenir un lecteur et un générateur d'écrivains (`readb` et `writerb`). Le serveur `b` est avertit de l'existence du lecteur via la fonction `connect` et transmet le générateur d'écrivains via le tuyau temporaire. La fonction `trans`, une fois la procédure transmise, sait qu'elle peut lire, sur le tuyau temporaire, un générateur d'écrivains, ce générateur est activé pour obtenir l'écrivain final : `((read ()) ())`.

Si la fonction `get` fait que le serveur `a` coincide avec le serveur `b`, i.e. `get = fun a f -> (f a)`, alors `trans` se comporte comme un duplicateur de tuyau. Néanmoins, cela permet de spécifier le protocole du tuyau dupliqué.

Avec cette notion de symétrie et de transitivité, en y ajoutant une réfléxivité évidente, la relation *connaître via un écrivain de fonctions* est une classe d'équivalence. Ainsi, on a la bonne intuition que tout ce qui peut être atteignable par le réseau peut se connecter directement par un tuyau typé.

Nous avons montré, qu'à partir de tuyaux typés, il est possible de créer d'autres tuyaux typés. Il reste à initialiser le processus en générant un premier tuyau. Nous allons donc rentrer un peu plus dans les détails des fonctions `publishPinet` et `lookupPinet` introduites dans notre exemple.

5. Le grain est livré

Comment un client peut-il entrer en communication avec un serveur s'ils ne sont pas dans la même classe d'équivalence vue précédemment ? La première solution, hautement non typable, est la

convention : "**on** *sait que sur la machine* **www.inria.fr** *il existe un lecteur de type* **TCP** *sur le port* **80** *lisant du* **HTTP**". Ici *HTTP* fait référence aux types de données qui vont circuler dans le tuyau, *TCP* et *80* sont plutôt en adéquation avec la structure du tuyau. Il y a la couleur des tuyaux et la couleur du liquide qui circule dedans. En regardant le fichier `/etc/services`, on serait tenté de croire qu'il existe une relation entre les deux couleurs. Mais ce n'est qu'une convention, de fait on peut utiliser le port 80 pour tenter de passer au travers d'un pare-feux et de faire transiter autre chose que du HTTP. Néanmoins, l'intérêt de cette convention est que l'Inria n'a pas à prévenir le monde entier qu'un serveur http a été installé sur sa machine.

Une solution, moins directe, consiste à faire intervenir une tierce personne pour établir la communication. Le serveur va publier, sur un site particulier, via un annuaire, la couleur de son tuyau d'entrée et la couleur de ce qu'il faut mettre dedans. La couleur du contenu peut-être implicite (annuaire mono-thématique comme dans le cadre de BitTorrent par exemple), seul l'adresse où se trouve le lecteur est importante (i.e. *www.inria.fr*), mais dans ce cas les tuyaux sont monomorphes et ne posent pas problèmes. Le client, de son coté, utilise l'annuaire pour construire l'écrivain avec les bonnes couleurs. La publication et la recherche se font au travers d'une clé, on publie un objet sous une certaine clé et on recherche un objet ayant été publié sous une certaine clé. Une particurité de Pinet est qu'il y a une relation de type, donc statiquement vérifiable, entre la clé et son contenu, le type Java de l'annuaire est :

```
Hashtable<Class<T>, CircList<OutFlowGenerator<Proc1<T>>>>
```

En Caml ce devrait être : $\texttt{TRepr}(\alpha) \rightarrow List(unit \rightarrow \alpha \rightarrow unit)$. Le fait que la liste soit circulaire est juste une optimisation pour assurer que les serveurs sont répartis de manière uniforme au fur et à mesure des requêtes. Il est important de remarquer que le type Java `OutFlowGenerator<Proc1<T>>` est proche du type du résultat de la fonction `lookupPinet` utilisée dans l'exemple du début de l'article. En fait, cette fonction va chercher un générateur d'écrivains sur le dictionnaire distant et active localement ce générateur pour produire l'écrivain.

Mais, comment font le client et le serveur pour rentrer en contact avec le propriétaire de l'annuaire ? La première solution, hautement non typable, est la convention : "**on** *sait que sur la machine* **ariwheels.inria.fr** *il existe un lecteur de type* **TCP** *lisant sur le port* **5359** *lisant des fonctions Java ...*" etc...etc.

Doit-on croire que le problème a juste été repoussé et que l'on est en face du dilemme de la poule et de l'oeuf ? En fait, oui. De la même manière qu'un vérificateur de type ne pourra jamais prouver que les arguments de la fonction `main` sont bien des chaines de caractères. Il faut accepter que tout programme démarre avec un ensemble, potentiellement vide si hors-réseau, de tuyaux typés. Néanmoins, à chaque fois que l'on repousse la création du tuyau originel, le type des valeurs devant circuler dans ce tuyau devient moins général.

6. Conclusion

Nous avons présenté le noyau de la librairie Pinet permettant de typer statiquement les valeurs circulant sur un réseau. Nous avons montré que cette propriété a été atteinte en admettant qu'il est possible de faire migrer du code, i.e. que les fonctions pouvaient transiter d'une machine à l'autre. Java a cette particularité, c'est une nécessité du fait que la principale entité que traite Java est l'objet, et sachant qu'un objet est l'encapsulation de données et de fonctions, i.e. une fermeture avec plusieurs points d'entrée, ne pas savoir envoyer une fonction sur le réseau reviendrait à reconnaître ne pas savoir envoyer des objets sur le réseau.

La notion de tuyaux typés est validée par le fait que, sur l'ensemble du code de la librairie, ne subsiste plus qu'un seul test de type dynamique. Il se trouve dans l'implémentation des lecteurs liés à un

flux de caractères, la classe `InputStream` de Java, et vérifie le résultat d'une dé-sérialisation (méthode `readObject` de Java correspondant à la fonction `Marshal.from_channel` d'Objective Caml).

On peut remarquer que ce test de type dynamique peut être supprimé en utilisant les travaux autour de la dé-sérialisation de Caml [6][7][2]. Il est possible d'aller encore plus loin en proposant de nouvelles règles de typage [5], mais le but de cet article est d'aborder le problème de la communication bien typée en utilisant un langage existant, Caml ou Java, sans introduire de nouvelles règles.

Pinet utilise de manière intense le polymorphisme générique hérité de Pizza[8]. En nous ramenant à des types fonctionnels nous avons pu exprimer les générateurs de tuyaux en Caml. Sans considérer la concision gagnée, cela nous a permis d'observer que les fonctions écrites originellement en Java étaient plus générales que prévu. Mais il aurait été fastidieux d'avoir à écrire ces types polymorphes en Java. Une des conclusions est donc que le fait d'obliger le programmeur à typer ses variables peut le conduire à minimiser la puissance des fonctions qu'il écrit.

Avoir des tuyaux typés implique qu'ils sont homogènes, i.e. que l'on ne peut pas écrire, par exemple, un entier puis un caractère. Ceci peut amener à créer plus de tuyaux que nécessaire. La structure d'un tuyau TCP est assez coûteuse à mettre en place. Dans ce contexte, le projet LogNet s'intéresse à une notion de *canal*, structure physique permettant une communication entre deux machines, pouvant contenir des structures plus légères, les *canaux virtuels*. Les tuyaux que nous avons décrits dans cet article peuvent être considérés comme des canaux virtuels. Il nous reste encore à essayer de typer les canaux.

Si au commencement de l'élaboration de la librairie Pinet, le sûreté par le typage était anecdotique et relevait du jeu : il aurait été certainement plus facile de rajouter des tests de type dynamiques là où le compilateur le demandait. *In fine*, l'effort fourni pour éliminer ces tests de type dynamiques nous semble profitable et nous a permis de structurer la librairie de manière plus abstraite. Enfin, cela nous a amené à des reflexions que nous avons essayé de partager dans cet article.

Références

[1] Didier Benza, Michel Cosnard, Luigi Liquori, and Marc Vesin. Arigatoni : A simple programmable overlay network. In *JVA '06 : Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing*, pages 82–91, Washington, DC, USA, 2006. IEEE Computer Society.

[2] John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for ocaml. In *ML '06 : Proceedings of the 2006 workshop on ML*, pages 20–31, New York, NY, USA, 2006. ACM.

[3] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1) :39–59, 1984.

[4] Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. Un modèle abstrait de composants adaptables. *revue TSI, Composants et adaptabilité*, 23(2), 2004.

[5] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ml for the join-calculus. In *CONCUR '97 : Proceedings of the 8th International Conference on Concurrency Theory*, pages 196–212, London, UK, 1997. Springer-Verlag.

[6] J. Furuse and P. Weis. Entrées/sorties de valeurs en caml. *Journées francophones des langages applicatifs*, pages 79–98, January 2000.

[7] Grégoire Henry, Michel Mauny, and Emmanuel Chailloux. Typer la dé-sérialisation sans sérialiser les types. *Journées francophones des langages applicatifs*, pages 133–146, January 2006.

[8] Martin Odersky and Philip Wadler. Pizza into java : Translating theory into practice. In *In Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM, 1997.

[9] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4) :149–160, 2001.

[10] Sun. Java remote method invocation - distributed computing for java. white paper, 1998. `http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp`

Foncteurs impératifs et composés: la notion de projets dans Frama-C

Julien Signoles[1]

1: CEA, LIST, Laboratoire de Sûreté des Logiciels
91191 Gif-sur-Yvette Cedex
`Julien.Signoles@cea.fr`

Résumé

Cet article présente la bibliothèque de projets de Frama-C, une plateforme facilitant le développement d'analyseurs statiques de programmes C. Grâce à sa description, nous présentons une utilisation originale des foncteurs du système de modules de ML qui exploite aussi bien leur caractère impératif que compositionnel. Ceci est le seul véritable recours pour réaliser la fonctionalité souhaitée de manière bien typée. En outre, nous montrons un exemple peu fréquent d'un même foncteur appliqué plusieurs centaines de fois. Cet article introduit aussi la plateforme Frama-C elle-même, à travers un de ses aspects essentiels, la notion de projet.

1. Introduction

Dans cet article, nous présentons une bibliothèque de projets intégrée à la plateforme *Frama-C* (*Framework for Modular Analysis of C*) [10] afin de permettre à un outil d'analyse statique de programmes C développé au sein de cette plateforme de travailler sur plusieurs programmes en parallèle de manière sûre, efficace et transparente pour le développeur. Cette bibliothèque utilise de façon fondamentale les modules paramétrés, plus connus sous le nom de foncteurs, qui sont les fonctions du système de modules intégré au langage *Objective Caml* (*OCaml*) [18].

Ce système de modules est un langage fonctionnel indépendant du langage de base sous-jacent [17] et pouvant être intégralement supprimé statiquement [23]. Il a été introduit dans le langage de programmation *OCaml* en 1996 et se fonde sur un modèle théorique clairement défini [14, 15, 16]. Même s'il était initialement peu utilisé, son usage s'est répandu au cours de la première moitié des années 2000 comme en attestent plusieurs articles sur le sujet [4, 5, 6, 17, 22, 24]. Il est maintenant largement utilisé. À quoi bon, alors, un nouvel article sur ce sujet ?

Outre la présentation d'une nouvelle bibliothèque, notre contribution est ici quadruple.

- D'abord, nous utilisons les foncteurs de manière originale : d'une part, leurs applications ont pour but principal d'effectuer des effets de bord et, d'autre part, leurs signatures permettent de les composer facilement, ce qui explique le titre «foncteurs impératifs et composés».
- Le deuxième apport est de montrer un cas précis dans lequel il n'y a pas moyen d'implémenter simplement les fonctionnalités souhaitées de manière bien typée à l'aide des autres traits du langage *OCaml* (programmation orientée objets et polymorphisme notamment) alors que, le plus souvent, il est possible de proposer une autre solution assez facilement quoiqu'éventuellement moins élégamment.
- La troisième contribution consiste à présenter un exemple concret dans lequel un même foncteur est très massivement appliqué (222 applications). À notre connaissance, il n'existe pas d'autres exemples du même type : même si certains codes sont amplement fonctorisés, le nombre de fois qu'un même foncteur est appliqué reste faible.

– Enfin, cet article présente la plateforme *Frama-C* au travers d'un de ses aspects essentiels, la notion de projet, ce qui n'a encore jamais été réalisé.

Plan Nous commencerons, section 2, par présenter la raison d'être de la bibliothèque de projets dans *Frama-C*. Nous entrerons ensuite dans le cœur de l'article, section 3, en présentant les fondements architecturaux de cette bibliothèque. Nous présenterons alors, section 4, son interface de haut-niveau avant d'aborder dans la section 5, les notions de sélections et de dépendances. Nous poursuivrons, section 6, en nous focalisant sur la sérialisation en présence de *hashconsing*. Le dernier point abordé, section 7, sera la séparation des notions d'états et de types de données.

2. La raison d'être de la bibliothèque de projets dans *Frama-C*

Dans cette section, nous introduisons de manière succincte le cadre dans lequel s'intègre la bibliothèque de projets à l'étude dans cet article. Ce cadre est la plateforme *Frama-C*. Néanmoins, le but ici *n'*est absolument *pas* de faire un exposé exhaustif des possibilités offertes par cette plateforme ; il est plutôt de présenter ce qu'il est nécessaire de connaître afin de bien comprendre la raison d'être de cette bibliothèque et les contraintes qui lui sont inhérentes. Pour les lecteurs plus amplement intéressés par *Frama-C*, une documentation à destination des utilisateurs [9] ainsi qu'un manual ciblant les développeurs [25] sont librement accessibles sur son site internet [10].

Frama-C (*Framework for modular analysis of C*) [10] est une plateforme logicielle *open source* extensible facilitant le développement d'analyseurs statiques collaboratifs de programmes C. Elle est entièrement développée en *OCaml* [18]. Outre proposer un puissant outil d'analyse statique aux fonctionalités multiples, utilisables par des acteurs industriels pour améliorer la fiabilité des logiciels écrits en *C* (en particulier les logiciels embarqués critiques de taille conséquente) — ce qui est son intérêt premier, *Frama-C* est aussi une importante bibliothèque *OCaml*[1] facilitant grandement aussi bien le prototypage rapide que le développement abouti d'analyseurs statiques dans ce langage.

Pour atteindre ce but, *Frama-C* est organisé autour d'une architecture logicielle à greffons similaire dans son esprit à celle de la plateforme Eclipse [11]. Les différents analyseurs sont intégrés à la plateforme comme autant de greffons, ces derniers pouvant échanger des informations permettant de faire collaborer les analyseurs entre eux. Ainsi par exemple, une analyse peut-elle exploiter les résultats d'une autre. En outre, ces différents greffons ont accès à un ensemble de fonctionalités commun fourni par le noyau de *Frama-C*. La première d'entre elles est la génération d'un arbre de syntaxe abstraite (AST) représentant le programme *C* à analyser, éventuellement étendu avec des annotations logiques écrites dans le langage *ACSL* (*ANSI C Specification Language*) [1].

Par ailleurs, cet AST est modifiable en place pour rendre sa construction plus efficace. Ensuite, toujours pour des raisons d'efficacité, des techniques de *hashconsing* [12, 13] et de mémoisation [19, 20] sont intensivement utilisées, requérant la présence d'états globaux mutables. De plus, le fait que *Frama-C* soit une plateforme extensible d'analyseurs collaboratifs oblige à avoir un certain nombre de tables globales extensibles pour enregistrer des informations émanant des greffons. Pour ces raisons, *Frama-C* utilise de nombreuses structures de données impératives (tables de hachage notamment) et possède un état global mutable de taille importante qui dépend du programme (et donc de l'AST) analysé.

De plus, *Frama-C* offre la possibilité de travailler sur plusieurs ASTs en parallèle. Un utilisateur peut exploiter ce parallélisme dans une étude de cas. Par exemple, *Frama-C* offre un outil de *slicing* [27, 28] qui permet de supprimer les parties non pertinentes — et uniquement celles-ci — d'un programme *C* par rapport à l'étude d'un certain critère, tout en conservant un programme compilable[2] : le *slicer* de *Frama-C* engendre un nouvel AST (plus petit que l'AST initial) que d'autres

1. La version distribuée de *Frama-C* comprend plus de cent mille lignes de code *OCaml*.
2. En réalité, le problème étant indécidable, il ne supprime qu'une sous-approximation des parties non pertinentes du programme afin d'être correct.

greffons peuvent analyser avec des chances de succès plus importantes qu'en analysant directement le programme initial (par exemple, dans le cas d'un analyseur par interprétation abstraite [7], le taux de faux positifs peut être moindre).

Plutôt qu'engendrer un nouvel arbre, il aurait été *a priori* possible de modifier l'AST en place. Cependant, la bonne fondation de ce dernier et la cohérence de certaines structures de données internes à *Frama-C* reposent sur des invariants complexes qu'une modification en place de l'AST brise. Plus généralement, générer de nouveaux arbres et cloisonner les informations associées à chacun d'eux réduient les risques d'incohérences et de bogues.

Le fait néanmoins que *Frama-C* possède plusieurs ASTs sur lesquels les greffons doivent travailler tend *a priori* inévitablement à rendre ardue la programmation de ces derniers. Il n'en est cependant rien grâce à la bibliothèque de projets dont la tâche principale est de permettre aux développeurs de *Frama-C*, et en particulier aux développeurs de greffons, de s'abstraire de cette vision multi-ASTs et d'effectuer l'essentiel du développement dans un monde à AST unique.

3. Projets et états

Un *projet* regroupe un AST et l'ensemble des états de *Frama-C* qui s'y rattachent, incluant ceux des greffons. Étant donné l'aspect multi-ASTs de *Frama-C*, plusieurs projets peuvent coexister dans *Frama-C*. Ces projets sont gérés par la bibliothèque que nous décrivons dans cet article. Cette dernière doit permettre l'accès aux différents états des différents projets, tout en rendant le plus transparent possible aux développeurs le fait que les états qu'ils définissent sont aussi nombreux que les ASTs. Ainsi par exemple, si un état est concrètement implanté par une table de hachage indexée par des instructions du programme, l'interface permettant d'accéder (en lecture ou en écriture) à cet état doit être celle d'une table de hachage usuelle, sans rendre visible le fait qu'elle dépend d'un AST particulier. En outre, d'une part, l'utilisation de la bibliothèque doit engendrer un faible surcoût à l'exécution étant donné que les accès aux états par les analyseurs peuvent être innombrables et, d'autre part, elle doit conserver les mêmes garanties de sûreté que celles apportées par le typage statique fort d'*OCaml*.

Ainsi, toute solution naïve fondée sur une table de hachage globale permettant de retrouver la valeur d'un état donné correspondant à un AST donné ne fonctionne pas dans la mesure où elle ne remplit aucune des trois conditions ci-dessus :.

- D'abord le développeur doit avoir conscience en permanence de la présence de plusieurs ASTs et doit même, dans l'absolu, paramétrer tous ses algorithmes par l'AST sur lequel il s'applique pour pouvoir effectuer une recherche dans la table de hachage.
- Ensuite le fait que les états soient de types différents (tables de hachage et références diverses, compteurs, etc) implique que la table soit une structure hétérogène, ce qui n'est pas typable en *OCaml*, sauf éventuellement en déployant des solutions lourdes à mettre en œuvre.
- Enfin, l'indirection engendrée par la table de hachage dès qu'on accède à un état induit un surcoût excessif à l'exécution. En effet, l'introduction d'une version préliminaire de la bibliothèque proche de celle décrite ici [3] a provoqué des pertes de performance sur certains exemples.

Il convient donc de trouver une solution plus astucieuse afin de pallier ces défauts.

3.1. Mécanisme général

La bibliothèque de projets entretient avec les modules qui l'utilisent une relation de type client-serveur avec *broadcast* qui est présentée de manière schématique figure 1. Ainsi, la bibliothèque de projets joue le rôle d'un serveur qui tient à jour l'ensemble des projets et conserve, pour chacun d'eux, la valeur de chaque état. Il existe en outre une notion de projet courant que chacun utilise de manière

3. Cette solution ne souffrait en particulier pas du problème de non-transparence et que partiellement du problème des structures hétérogènes.

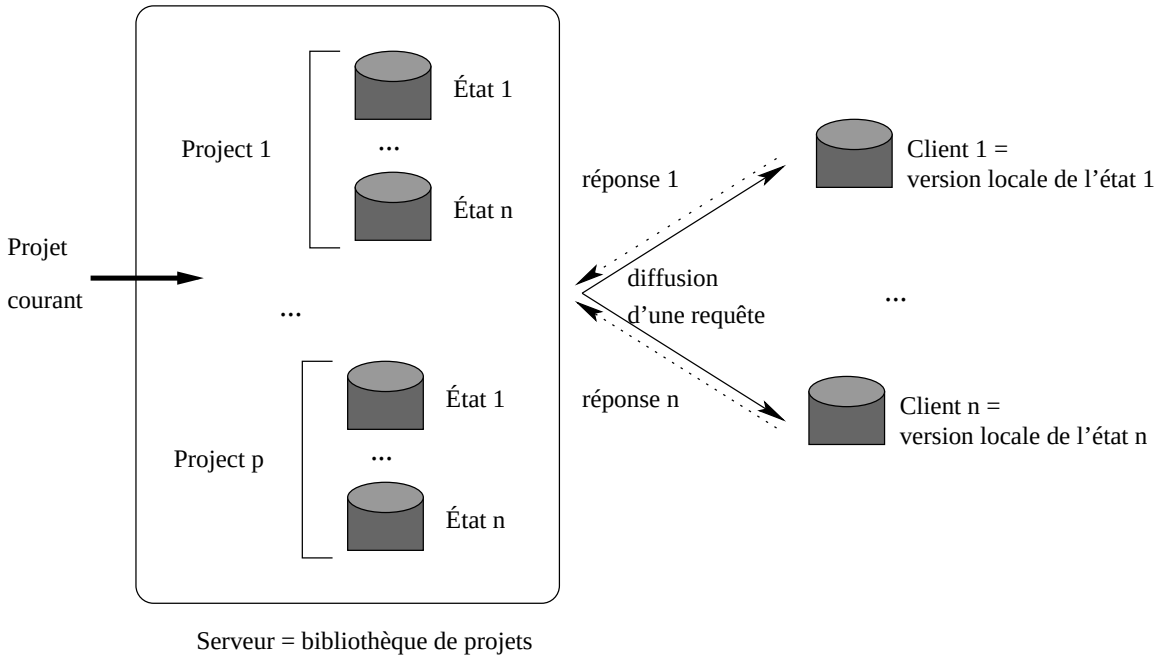


FIGURE 1 – Fonctionnement de type client-serveur avec *broadcast*.

transparente : chaque module définissant un état en possède une version locale, le client, sur laquelle il travaille au quotidien. Ce principe est similaire à celui d'un gestionnaire de versions comme *CVS* (*Concurrent Versions System*) [26]. Il en diffère cependant de manière fondamentale dans son mode de synchronisation. En effet, contrairement à *CVS* dans lequel le client décide du moment où il se synchronise avec le serveur, c'est ici le serveur (*i.e.* la bibliothèque) qui décide du moment où un client (*i.e.* la version locale d'un état) doit être synchronisé avec lui. Pour cela, il diffuse (*broadcasts*) des requêtes à chacun d'eux. De cette manière, la bibliothèque garantit la cohérence entre le projet par défaut et les versions locales de ses clients. Ainsi notamment, à chaque fois que le projet courant change, la bibliothèque se met à jour avec les données locales de chaque client de manière à sauvegarder les données associées à l'ancien projet courant (opération correspondant à un `cvs commit`) puis change les données locales de chaque client afin qu'elles correspondent à celles associées, sur le serveur, au nouveau projet courant (opération correspondant à un `cvs update`).

3.2. Enregistrement d'un nouvel état

Le point essentiel sur lequel repose le mécanisme précédemment décrit est celui d'*enregistrement* des états car il suppose que chaque état soit connu de la bibliothèque afin d'en être un de ses clients. Ce mécanisme nécessite en particulier que la bibliothèque soit capable d'obtenir un état local au moment d'un *commit* et de le modifier au moment d'un *update*. En outre, il fait l'hypothèse que la bibliothèque est capable d'associer à chaque projet l'ensemble des valeurs des états (qui sont, rappelons-le, de types différents) au moment de la dernière synchronisation, ce qui semble requérir l'utilisation d'une structure de données hétérogène, qui nuirait à la sûreté du mécanisme. Nous allons voir qu'il n'en est rien.

L'idée principale ici est d'utiliser un foncteur `Register` pour enregistrer les nouveaux états : le but principal de son application est de faire connaître chaque nouvel état à la bibliothèque de projets. Ce

foncteur est paramétré par un état de signature `INPUT` définie de la façon suivante.

```
module type INPUT = sig
  type t                    (* type de l'état *)
  val create: unit → t      (* comment créer un nouvel état *)
  val get: unit → t         (* comment accéder à l'état local *)
  val set: t → unit         (* comment remplacer l'état local par un nouveau *)
end
```

Cette signature permet à l'utilisateur de définir le type de l'état à enregistrer ainsi que les moyens d'y accéder en lecture (`get`) et en écriture (`set`). En outre, la bibliothèque a besoin de pouvoir créer un nouvel état (`create`) lors de la création d'un nouveau projet. Ces fonctions vont permettre à la bibliothèque de communiquer avec la version locale de l'état enregistré de cette manière.

Dans ce but, le corps du foncteur contient une table associant à chaque projet la valeur de cet état. On peut noter que cette table est une structure de données homogène car elle ne concerne qu'un seul état. Avec son aide, il est facile de définir les opérations `commit` et `update` précédemment introduites ainsi que la requête de création `create` comme le montre le code suivant.

```
module Register(State:INPUT) = struct
  type t = { mutable state: State.t }
  (* table associant à chaque projet représenté par un entier, la valeur de l'état
     au moment de la dernière synchronisation *)
  let tbl: (int, t) Hashtbl.t = Hashtbl.create 17
  (* fonction auxiliaire *)
  let find p = try Hashtbl.find tbl p with Not_found -> assert false
  (* requêtes *)
  let commit p = if is_current p then (find p).state ← State.get ()
  let update p = if is_current p then State.set (find p).state
  let create p =
    assert (not (Hashtbl.mem tbl p));
    Hashtbl.add tbl p { state = State.create () };
    update p
end
```

Le type `t` introduit ici est isomorphe au type `ref` des références d'*OCaml*. S'il est défini, c'est qu'il est en réalité étendu avec un autre champ que nous ne détaillerons pas ici [4]. De plus, nous utilisons le fait qu'un projet est représenté par un entier, ce qui sera justifié section 3.3. Nous n'effectuons aussi que les synchronisations entre l'état client et le projet courant grâce à la fonction `is_current`, qui sera présentée section 3.3. Par ailleurs, le corps du foncteur vérifie également l'invariant que la fonction `create` est appelée une et une seule fois par projet.

Le code ci-dessus n'est cependant pas suffisant car il ne s'applique qu'à un seul état. En constatant que les types des fonctions en jeu sont indépendants des types des états, nous introduisons un type `state_operations` et un module `States` permettant de diffuser les requêtes à l'ensemble des clients.

```
(* type des requêtes à diffuser *)
type state_operations =
  { create: int → unit; commit: int → unit; update: int → unit }
module States = struct
   (* ensemble des requêtes des clients *)
```

4. De manière générale, dans un but didactique, le code présenté ici est simplifié par rapport à l'implémentation réelle. Néanmoins, seules des fonctionalités secondaires ou des optimisations de la bibliothèque ne sont pas introduites.

```
  let states : state_operations list ref = ref []
  (* diffusion des différentes requêtes *)
  let create p = List.iter (fun s → s.create p) !states
  let commit p = List.iter (fun s → s.commit p) !states
  let update p = List.iter (fun s → s.update p) !states
  (* enregistrement d'un nouveau client *)
  let register s = s :: !states
end
```

Désormais, il ne nous reste plus qu'à enregistrer les requêtes de chaque client à l'aide de la fonction `States.register`. Pour cela, dans le corps du foncteur `Register`, nous effectuons un effet de bord qui enregistre automatiquement chaque module client dès lors qu'il est appliqué à ce foncteur.

```
module Register(State:INPUT) = struct
  (* corps du foncteur précédemment écrit étendu par *)
  let self = { create = create; commit = commit; update = update }
  let () = States.register self
end
```

Du point de vue utilisateur, seul ce dernier effet de bord est important quand il enregistre un nouvel état. Ainsi, la signature `OUTPUT` du foncteur `Register` peut elle être vide.

```
module type OUTPUT = sig end
module Register(State:INPUT) : OUTPUT
```

Nous appelons «foncteur impératif» un foncteur dont le principal intérêt réside dans l'effet de bord ayant lieu au moment de ses applications, tel le foncteur `Register`. Nous ne connaissons à ce jour aucun autre code utilisant de tels foncteurs.

Par ailleurs, à cause de la table de hachage interne à `Register` dont le type `(int, State.t) Hashtbl.t` dépend du paramètre du foncteur, et plus généralement de l'état à enregistrer, toute autre solution utilisant d'autres traits du langage *OCaml* est vouée à l'échec. En effet, les deux solutions alternatives habituelles aux modules paramétrés sont les fonctions polymorphes et les objets. Concernant les premières, aucune solution ne peut fonctionner à cause des types polymorphes non généralisables d'*OCaml* qui empêcherait de typer la table de hachage, tandis que, pour les objets, nous serions limités par le fait que le type de l'état doit être connu (et utiliser des objets polymorphes aboutirait au même problème qu'utiliser des fonctions polymorphes).

On peut cependant remarquer une analogie entre notre solution et la programmation objet : l'enregistrement `self` contenant les requêtes de l'état enregistré peut être assimilé à un objet contenant les méthodes `create`, `commit` et `update`. Ces dernières ne nécessitent pas de prendre `self` en argument car elles lui sont automatiquement appliquées. Autrement dit, l'application de notre foncteur impératif est analogue à la création d'un nouvel objet dont l'utilisation serait interne à la bibliothèque de projets. Le nom `self` a ainsi été choisi dans le but de mettre en évidence cette analogie.

3.3. Opérations de base sur les projets

Maintenant que nous avons défini le module `States` permettant de diffuser des requêtes à chaque client, nous sommes en mesure de définir des opérations sur les projets proprement dits, au sein du module `Project`.

Pour représenter l'ensemble des projets coexistants, nous utilisons une file similaire à celle fournie par le module `Queue` d'*OCaml* mais étendue avec certaines opérations permettant d'accéder (en temps au pire linéaire) aux éléments qui ne sont pas en tête de file. Le module, appelé `Q` et définissant ette structure de données ne présente pas d'intérêt majeur : il n'est pas décrit dans cet article.

L'ensemble des projets de *Frama-C* est créé de la façon suivante.

```
let projects = Q.create ()
```

En considérant que le projet courant est toujours en tête de la file et que le type des projets est `int`[5], les opérations sur le projet courant sont immédiates à implémenter.

```
let current () = Q.peek projects
let is_current p = p = current ()
```

La création d'un nouveau projet diffuse la requête de création à chaque client pour que le nouveau projet contienne une instance de chaque nouvel état. Un compteur est en outre utilisé afin de garantir l'unicité de chaque projet.

```
let create =
  let cpt = ref 0 in
  fun () →
    if cpt = -1 then invalid_arg "cannot create project: too many projects";
    incr cpt;
    let p = !cpt in
    Q.add p projects; (* le nouveau projet est ajouté en queue de la file des projets *)
    States.create p;
    p
```

À présent, l'opération de changement de projet courant suit l'algorithme précédemment décrit : après avoir vérifié que le projet est bien enregistré, le serveur se synchronise avec ses clients afin, en premier lieu, d'enregistrer les modifications dans l'ancien projet courant puis, en second lieu, de changer la valeur de chaque client. Entre-temps, on effectue le changement de projet courant.

```
let set_current p =
  if not (Q.mem p projects) then invalid_arg "set_current";
  States.commit (current ()); (* sauve sur le serveur les modifications effectuées en local *)
  Q.move_at_top p projects;   (* p est maintenant le projet courant *)
  States.update p             (* change la valeur des états locaux *)
```

Le coeur de la bibliothèque est maintenant décrit. Un certain nombre d'autres opérations sont également fournies afin de permettre une meilleure utilisation des projets. Nous n'en détaillerons ici qu'une, la copie de projets. Cette fonction copie par défaut le projet courant.

```
let copy ?(src=current()) dst =
  States.commit src;
  States.copy src dst;
  States.update dst
```

Afin de garantir que les bonnes valeurs des états sont copiées de la source vers la destination, une synchronisation avec le serveur (qui n'est effectuée, rappelons-le, que dans le cas où la source est le projet courant) est nécessaire avant la copie. De même, une mise à jour du projet `dst` est effectuée après la copie. Par ailleurs, de manière similaire aux autres, une requête de copie est effectuée afin d'être en mesure de copier un état d'un projet vers un autre. Dans ce but, la signature `INPUT` du paramètre du foncteur `Register` est étendue avec une fonction `copy : t → t` qui suppose que l'on est capable d'effectuer une copie profonde d'un état. Nous y reviendrons dans les sections 5 et 7.

5. En réalité, le type des projets est un enregistrement dont un des champs est un identifiant à valeur entière. Les autres champs sont relatifs aux noms des projets. Ne présentant que peu d'intérêt, ces derniers sont ignorés dans cet article et les projets sont assimilés à des entiers.

3.4. Cloisonnement des projets

Montrons à présent comment utiliser la bibliothèque et le foncteur `Register` pour créer un état contenant un entier. Habituellement, un tel état est représenté par une référence sur un entier et, donc, on peut s'attendre à enregistrer l'état de la façon suivante.

```
let create () = ref 0
let state = create () (* la version locale de l'état *)
include Register
  (struct
     type t = int ref
     let create = create
     let get () = state
     let set s = state := !s
     let copy s = ref !s (* copie profonde *)
   end)
```

Ici, la primitive `include` est utilisée pour ne pas polluer inutilement l'espace des noms de module : la structure résultante de l'application du foncteur `Register`, rappelons-le, est vide et donc inutile.

La solution ci-dessus ne fonctionne cependant pas. En effet, la référence `state` correspondant à la version locale de l'état serait alors partagée entre tous les projets à cause du code de la fonction `get`. Dès lors, par *aliasing*, toutes les valeurs de cet état seraient identiques dans tous les projets. Ce n'est certainement pas ce qui est souhaité. Pour la rendre correcte, il faut modifier le code de la fonction `get` afin d'empêcher tout *aliasing*, ce que fait la ligne suivante.

```
let get () = ref !state
```

Même si, ici, le code de `get` est proche du code de `copy`, il n'en est rien dans le cas général car `copy` doit effectuer une copie profonde de son argument, tandis que `get` se contente de retourner une copie du pointeur de tête.

On peut remarquer que cette solution n'engendre aucun surcoût à l'exécution pour l'utilisateur de l'état car on y accède directement *via* `state`, et donc aussi rapidement qu'en l'absence de projets. Rien n'est cependant gratuit : les opérations de projets qui émettent des requêtes en paient le prix, car ces émissions sont linéaires par rapport au nombre d'états. Ainsi, les algorithmes qui effectuent beaucoup de changement de projets sont un peu plus coûteux mais, d'une part, ils ne sont pas si nombreux et, d'autre part, nous verrons section 5 qu'il est possible de limiter le nombre de requêtes émises. En pratique, les opérations de manipulation de projets *via* l'interface graphique de *Frama-C* sont instantanées, tandis que les algorithmes de transformation de programmes comme le *slicing* sont les seuls à payer un léger surcoût qui, en outre, est plus faible quand le programme analysé est plus gros (la complexité de ces algorithmes est fonction de la taille du programme analysé).

Une telle implémentation est néanmoins source de bogues et quelque peu fastidieuse à écrire. Pour éviter que l'utilisateur ait à la développer, des modules de plus haut niveau sont fournis par la bibliothèque de projets. Ils seront présentés dans la section 4.

De manière plus générale, les fonctions fournies à `Register` doivent vérifier les trois propriétés suivantes afin d'assurer que l'état ainsi enregistré n'est pas partagé entre les différents projets [6].

$$\texttt{create ()} \text{ retourne une valeur fraîche} \tag{1}$$

$$\forall \text{ valeur } v, \texttt{copy } v \text{ retourne une valeur fraîche} \tag{2}$$

$$\forall \text{ valeurs } v_1, v_2 \text{ telles que } v_1 \texttt{ != } v_2, \texttt{set } v_1 \texttt{ ; get () != } v_2 \tag{3}$$

6. Nous utilisons dans cet article les mêmes notations que celles d'*OCaml* pour distinguer égalités physique et structurelle. Par ailleurs, une autre propriété mettant en jeu la fonction `clear` non mentionnée dans cet article est également nécessaire.

La première propriété assure l'absence d'*aliasing* avec toute nouvelle valeur. La deuxième assure la même chose pour toute valeur issue d'une copie. La dernière propriété est un critère d'indépendance local qui apporte la garantie que modifier la version locale d'un état ne modifie pas une autre version de l'état par *aliasing*. C'est cette dernière propriété qui n'était pas vérifiée dans notre première tentative d'application du foncteur `Register`. Ces spécifications des opérations sont relativement faibles mais sont suffisantes pour garantir la propriété visée. En effet, sous ces hypothèses, il est possible de montrer une propriété de cloisonnement des différents projets.

Propriété 1 (Cloisonnement des projets) *Si les hypothèses 1, 2 et 3 sont vérifiées par chaque argument appliqué au foncteur* `Register`*, alors aucune version d'un état enregistré en appliquant ce foncteur n'est partagée avec une autre version d'un état d'un autre projet, ce qui peut être formellement exprimé par :*

$$\forall \text{ projets } p_1, p_2 \text{ tels que } p_1 \texttt{ <> } p_2, \forall \text{ états } s_1, s_2, \texttt{(find}_{s_1}\ p_1\texttt{).state != (find}_{s_2}\ p_2\texttt{).state}.$$

Les fonctions $\texttt{find}_{s_i}$ *représentent ici les fonctions auxiliaires* `find` *obtenues en appliquant le foncteur* `Register` *pour créer l'état* s_i*.*

En réalité, la véritable propriété de cloisonnement que l'on souhaiterait avoir est qu'il n'y ait aucun pointeur accessible à partir d'un projet qui soit accessible à partir d'un autre (c'est-à-dire étendre la propriété 1 à la clôture transitive des accès pointeurs). Cette propriété n'est pas vérifiée par la bibliothèque sous les conditions énoncées. Pour qu'elle le soit, il faudrait que chaque client assure qu'aucun pointeur accessible à partir de `get ()` ne soit partagé, ce qui oblige à coder la fonction `get` par une copie profonde comme ci-dessous (en supposant que `copy` effectue une copie profonde).

```
let get () = copy !state
```

Cela n'est pas acceptable pour des raisons d'efficacité. Par conséquent, pour que cette propriété soit vérifiée, il est de la responsabilité de l'utilisateur de l'état (et non de celle de l'utilisateur du foncteur `Register`) de ne pas créer d'*aliasing* entre projets, ce qui peut notamment malencontreusement arriver lorsqu'on effectue des transformations de programmes qui, comme le *slicing*, nécessitent de copier des données d'un projet à un autre. Cette propriété est néanmoins vérifiée par *Frama-C* qui fournit des modules facilitant la programmation des analyseurs (en particulier, un visiteur permettant de copier l'AST en y apportant des modifications) [25].

4. Modules de plus haut-niveau

Comme nous venons de le voir, appliquer le foncteur `Register` requiert d'assurer des invariants sur les fonctions à fournir qui sont faciles à casser. Pour éviter, d'une part, l'introduction de bogues de cette nature et, d'autre part, l'écriture de codes fastidieux, la bibliothèque de projets fournit un ensemble de foncteurs de plus haut niveau d'abstraction, qu'il suffit d'appliquer afin d'enregistrer un état. Ainsi par exemple, il existe un foncteur `Ref` permettant d'enregistrer une référence. La signature de ce foncteur est définie de la manière suivante.

```
module type REF_INPUT = sig
  type t                    (* type de la valeur référencée *)
  val copy: t → t           (* copie profonde *)
  val default: unit → t     (* valeur par défaut stockée dans la référence *)
end

module type REF_OUTPUT = sig
```

```
  type data                (* type de la valeur référencée *)
  val get: unit → data (* Accès à la valeur référencée *)
  val set: data → unit (* Change la valeur référencée *)
end

module Ref(Data:REF_INPUT) : REF_OUTPUT with type data = Data.t
```

Nous ne détaillons pas ici l'implémentation qui reprend et généralise l'exemple de la référence donné précédemment. Cet exemple s'écrit maintenant en une seule ligne comme suit.

```
module Mon_État = Ref(struct type t = int let copy x = x let default () = 0 end)
```

En outre, d'un point de vue génie logiciel, cette surcouche de plus haut-niveau a un effet bénéfique supplémentaire à celui de masquer le foncteur `Register` : il permet de masquer la définition de l'état interne lui-même et de ne fournir, *via* la signature `REF_OUTPUT`, que des fonctions le manipulant, à savoir ici `get` et `set`. Elles correspondent aux opérateurs respectivement `(!)` et `(:=)` d'*OCaml*. En outre, le foncteur permet de créer des références de n'importe quel type, ce qui remplace ainsi le polymorphisme habituel associé aux références.

Sur le même principe, une série d'autres foncteurs est également fournie afin de faciliter l'enregistrement de tous les types de données fournis par la bibliothèque standard d'*OCaml*. Ainsi, dans l'ensemble de *Frama-C*, seuls les enregistrements de deux états nécessitent l'utilisation du foncteur `Register` de bas-niveau. *Frama-C* propose également un module facilitant l'enregistrement d'états utilisant des types de données de l'AST (par exemple, des tables indexées par des instructions *C*). Ce module lui-même n'utilise pas directement le foncteur de bas-niveau.

5. Sélections et dépendances d'états

De manière transverse au mécanisme de type client-serveur avec *broadcast* mis en place pour communiquer avec les versions locales des états, et au mécanisme d'enregistrement qui en découle, un système de sélections et de dépendances d'états a aussi été instauré.

Une *sélection* est un ensemble d'états qui permet de contrôler les destinataires des requêtes émises par le serveur. Ceci a deux intérêts : d'une part, cela permet d'optimiser certains traitements (par exemple, une opération temporaire de changement de projets) en effectuant moins de synchronisations avec les états locaux et, d'autre part, cela permet de contrôler plus finement les actions effectuées. Ainsi par exemple, une opération comme la copie (profonde, rappelons-le) de projets peut être coûteuse et fastidieuse à coder sur des états modifiables en place (passe linéaire sur la structure de données). En revanche, cette opération n'est utile que pour certains états : l'AST, par exemple, n'a jamais besoin d'être copié. L'utilisation des sélections dans ce contexte permet de choisir les états à copier d'un projet à un autre et de ne pas coder [7] les fonctions de copie qui ne sont jamais utilisées.

Afin de pouvoir créer de tels ensembles, il est nécessaire de pouvoir manipuler les états et, en particulier, de pouvoir les comparer. Dans ce but, le type `state_operations`, introduit page 41 et représentant le type des états, contient en réalité un nom (une chaîne de caractères), fourni au moment de l'application du foncteur `Register`, qui permet d'identifier les états de manière unique [8].

Ensuite, les valeurs `self` de ce type, engendrées par les applications du foncteur `Register`, peuvent être utilisées comme représentant d'un état. Afin que l'utilisateur puisse les manipuler dans les sélections, nous étendons la signature vide `OUTPUT` précédemment introduite avec `self`.

7. Comme la fonction `copy` est requise en argument des foncteurs, elle est en réalité codée comme **fun** _ → **assert** `false`.

8. Toute solution classique utilisant un compteur pour assurer l'unicité ne fonctionne pas dans la mesure où le nombre et l'ordre des états peuvent varier et, donc, la numérotation automatique n'attribuerait pas toujours les mêmes numéros aux mêmes états.

```
type state (* dans l'implémentation, égal à state_operations *)
module type OUTPUT = sig val self: state end
```

Afin de pouvoir contrôler l'émission des requêtes, les fonctions effectuant des *broadcasts* sont paramétrées avec deux sélections optionnelles `only` et `except`. Les états E auxquels les requêtes émises par ces fonctions parviennent sont les suivants (où $\mathcal{E}$ désigne l'ensemble des états enregistrés) :

$$E = \begin{cases} \mathcal{E} \setminus \texttt{except} & \text{si } \texttt{only} = \emptyset \\ \texttt{only} \setminus \texttt{except} & \text{sinon.} \end{cases}$$

Par exemple, la ligne de code suivante copie l'ensemble des états de *Frama-C*, à l'exception de son AST, du projet courant vers le projet `dst`. Ce dernier projet conserve donc un AST inchangé.

```
let () = copy ~except:(Selection.singleton Ast.self) dst
```

Si les sélections permettent un contrôle plus fin sur les diffusions effectuées par la bibliothèque de projets, elles ont cependant deux défauts : d'une part, il peut être pénible de créer des sélections avec beaucoup d'états et, d'autre part, il est facile de briser la cohérence de l'application en effectuant des sélections incorrectes. Ainsi par exemple, la ligne de code précédente copiant l'AST de *Frama-C* est fausse car elle plonge *Frama-C* dans un état incohérent. En effet, les états dépendants de l'AST (comme les résultats des analyses) sont copiés dans le projet `dst` alors que l'AST de ce projet demeure inchangé. Hors, les résultats des analyses copiés peuvent être incorrects pour cet AST.

Pour corriger ces défauts, une notion de *dépendance* entre états a été introduite et, à chaque introduction d'un état dans une sélection, on doit obligatoirement spécifier comment gérer ses dépendances à l'aide d'une valeur du type suivant.

```
type how = DoNotSelectDependencies | SelectDependencies | OnlySelectDependencies
```

Ainsi en réalité, l'exemple de la copie ci-dessus était mal typé et devrait être écrit comme suit afin de ne copier ni l'AST ni ses dépendances.

```
let () = copy ~except:(Selection.singleton Ast.self SelectDependencies) dst
```

Les dépendances entre états jouant un rôle clé dans la cohérence de l'application, un champ `dependencies` est présent dans la signature `INPUT` du foncteur `Register` pour contraindre le développeur à expliciter les dépendances de l'état qu'il enregistre. En interne, la bibliothèque de projets gère un graphe de dépendances *Ocamlgraph* [4, 5] qui remplace, à l'intérieur du module `States`, la référence vers la liste d'états précédemment codée. Nous ne détaillons ici ni cette implémentation, ni celle des sélections qui ne présentent pas d'intérêt majeur.

6. Sérialisation et hashconsing

De part son rôle même, la bibliothèque de projets connaît l'ensemble des états globaux du logiciel et, très exactement, la file de projets `projects` *est* cet état global. Ainsi, pour sauvegarder l'état de l'application sur le disque, il suffit de sérialiser cette valeur. C'est la raison pour laquelle, la bibliothèque de projets est responsable de la sérialisation [9] et de la désérialisation. Cette fonctionnalité est fondamentale pour *Frama-C* car les analyses peuvent nécessiter énormément de temps d'exécution (plusieurs heures, voire plusieurs jours). Il est donc indispensable de bénéficier d'un mécanisme permettant d'obtenir leurs résultats sans avoir à les exécuter de nouveau. Ce mécanisme est la

9. La sérialisation est aussi connu sous le nom de *marshalling* en anglais.

mémoisation [19, 20] au cours d'une session simple de *Frama-C* et la sérialisation lorsque *Frama-C* est quitté puis relancé.

La sérialisation consiste à écrire l'état global de *Frama-C* (*i.e.* la valeur `projects`) sur le disque, projet par projet et, pour chacun d'eux, état par état. La désérialisation suit le même procédé. Le fait que *Frama-C* soit une architecture à greffons rend le chargement des états d'un projet plus ardu car un greffon et les états qu'il définit peuvent être présents à la sauvegarde mais pas au chargement, et réciproquement. Une solution, non décrite ici mais qui résoud ce problème, a été déployée [21] en utilisant le nom des états.

Hormis les aspects propres à la sérialisation et à la désérialisation, ces fonctions procèdent de manière similaire aux autres : elles diffusent les requêtes de sérialisation/désérialisation aux états en étendant le type `state_operations` regroupant le type des requêtes des clients.

```
type state_on_disk = { value: Obj.t; typ: string }
type state_operations =
  { ... (* opérations précédemment données *)
    serialize: int → state_on_disk;
    unserialize: state_on_disk → int }
```

Comme nous pouvons le constater, ces opérations utilisent le module `Obj` d'*OCaml* qui n'apporte aucune garantie de sûreté. En réalité, les opérations de sérialisation et de désérialisation fournies par *OCaml* ne sont déjà pas sûres et seules des extensions du compilateur [2] apportent cette garantie. C'est la raison pour laquelle nous nous permettons ici l'utilisation de `Obj`. En outre, le champ `typ` est utilisé pour apporter plus de sûreté à la désérialisation, comme nous l'expliquerons section 7.

La difficulté majeure de la désérialisation est la bonne gestion du *hashconsing* [12, 13]. Cette fonctionalité permet de ne pas duppliquer les valeurs structurellement égales [10] et, ainsi, de gagner en efficacité, aussi bien en mémoire qu'en temps [3]. Pour l'implémenter, des tables globales — appelées tables de *hashconsing* — sont nécessaires pour assurer l'unicité des valeurs créées. Une telle implémentation a cependant deux inconvénients. La première est la difficulté d'empêcher des fuites mémoires [8] et la seconde est la difficulté de la sérialisation. Concernant ce second point en effet, deux tables de *hashconsing* différentes ont été utilisées pour créer les valeurs présentes sur le disque et celles présentes en mémoire. Par conséquent, la propriété de non-dupplication de valeurs structurellement égales est perdue, ce qui consomme inutilement de la mémoire et peut surtout entraîner des incorrections dans les analyseurs qui utilisent cette propriété. La solution mise en œuvre pour la restaurer est de rehâcher une et une seule fois chaque valeur pointée par une désérialisée afin de la rendre compatible avec les tables de *hashconsing* existantes. Nous ne détaillons pas ici cette solution mais elle nécessite que chaque client fournisse une fonction `rehash` de type `t → t`. Le surcoût engendré à l'exécution par ce rehachage des valeurs existe certes, mais la désérialisation demeure une opération rapide (instantanée en général et prenant quelques secondes pour charger des résultats d'analyse qui ont pris plusieurs jours à être calculés). La correction de la désérialisalisée en présence de *hashconsing* est à ce prix.

7. Types de données

Les fonctions `rehash` et `copy` requises pour enregistrer un nouvel état sont en réalité plutôt des opérations sur son type de données que sur l'état lui-même, comme le montre d'ailleurs leur type commun `t → t` : pour un type donné, quelque soit l'état de ce type enregistré, ces fonctions seront les mêmes. Pour cette raison, ces deux fonctions ont été enlevées de la signature `INPUT` et ajoutées comme argument au foncteur.

10. Pour *Frama-C*, il s'agit de ne pas duppliquer uniquement les valeurs structurellement égales *dans un même projet*, afin de respecter la propriété de cloisonnement précédemment énoncée.

```
module Datatype = struct
  module type INPUT = sig
    type t
    val copy: t → t
    val rehash: t → t
  end
  module type OUTPUT = sig include INPUT ... (* voir plus loin *) end
end
module Register(D:Datatype.INPUT)(S:INPUT with type t = D.t) : OUTPUT
```

Il revient donc à l'utilisateur de fournir ce nouvel argument `D`. L'intérêt de ce nouvel argument est la factorisation des implémentations des types de données. Pour cette raison, en parallèle des deux modules de haut-niveau proposant des foncteurs d'enregistrement d'états, il existe deux modules de haut-niveau proposant des modules pour les types de données prédéfinis dans la bibliothèque standard d'*OCaml* et dans *Frama-C*. Ainsi par exemple, sont définis la structure `Int` et le foncteur `List`.

```
module Int: Datatype.OUTPUT with type t = int
module List(D:Datatype.INPUT): Datatype.OUTPUT with type t = D.t list
```

De telles définitions permettent facilement de composer de nouveaux types de données. Ainsi, voici finalement comment enregistrer comme état une table de hachage, de taille initiale 17, associant à des entiers des listes d'entiers.

```
module Mon_État =
  State.Hashtbl (* foncteur prédéfini pour les tables de hachage *)
    (Datatype.Int) (* type des clés *)
    (Datatype.List(Datatype.Int)) (* types des valeurs associées aux clés *)
    (struct (* informations supplémentaires *)
       let dependencies = [] (* dépendances de cet état *)
       let name = "Mon_État" (* nom de l'état *)
       let size = 17         (* taille par défaut de la table de hachage *)
     end)
```

Par ailleurs, un mécanisme d'enregistrement des types de données a été mis en place dans le même esprit que celui des états. Ces deux mécanismes divergent cependant dans leur mode opératoire. Alors que, comme nous l'avons vu, l'enregistrement des états consiste fondamentalement à effectuer des effets de bord, l'enregistrement d'un type de donnée consiste principalement à ne rien faire (!) comme le montre le code qui suit.

```
module Datatype = struct module Register(D:INPUT) = D (* foncteur identité *) end
```

À quoi bon, alors, un tel enregistrement ? En réalité deux effets de bord sont aussi effectués lors de l'application du foncteur [11].

Le premier effet est de fournir des fonctions `compare`, `equal`, `hash` et `physical_hash` [12] prédéfinies qu'il est ensuite possible de facilement personnaliser. En particulier, les modules prédéfinis fournissent ainsi au développeur des versions optimisées (si possible) de ces fonctions, souvent critiques pour l'efficacité des analyseurs. La signature `Datatype.OUTPUT` obtenue en sortie de l'application des foncteurs est donc un sous-type de la signature `Datatype.INPUT` (*i.e.* contient en particulier au moins les mêmes éléments), ce qui n'empêche donc pas de composer les foncteurs.

11. Composer de tels foncteurs impératifs alors même que l'ordre d'évaluation des foncteurs est non spécifié ne pose pas de problème ici car les effets de bord effectués sont indépendants les uns des autres.

12. La fonction `hash` est une fonction de hachage qui doit être compatible avec l'égalité structurelle fournie par `equal`, tandis que `physical_hash` doit être une fonction de hachage compatible avec l'égalité physique `(==)`.

Le second effet est d'enregistrer un nom unique associé à chaque type pour offrir plus de sûreté à la désérialisation. Ce mécanisme de nommage est identique à celui des états. Il permet de vérifier qu'un état, sérialisé avec un type de nom x, est bien désérialisé avec un type du même nom. Ceci n'est pas fiable à 100% dans la mesure où il est théoriquement possible d'intervertir deux noms de type entre deux exécutions de *Frama-C* mais, en pratique, cette vérification s'avère suffisante et utile.

8. Conclusion

Nous avons présenté la bibliothèque de projets de *Frama-C* qui, à l'aide d'un mécanisme de type client-serveur avec *broadcast*, permet à tout développeur de cette plateforme de travailler sur plusieurs programmes *C* en parallèle de manière sûre, efficace et transparente. Pour l'implémenter, l'utilisation de foncteurs, dits «impératifs», dont le but principal est d'effectuer des effets de bord au moment de leurs applications est nécessaire, ce qui est original. D'ailleurs deux bogues du typeur d'*OCaml*[13] ont été trouvés en les utilisant. En outre, la composition de foncteurs est également requise pour faciliter l'utilisation de la bibliothèque. De plus, la présentation de *Frama-C* à travers un de ses aspects essentiels n'avait encore jamais été réalisée et constitue un exemple peu fréquent dans lequel un même foncteur est massivement appliqué (222 fois ici).

Le code montré dans cet article a été simplifié dans un but didactique. La version non simplifiée, 1600 lignes de code environ, peut être trouvée dans le répertoire `src/project` de *Frama-C* [10]. Elle implémente des fonctionalités secondaires et utilise un style de programmation très défensif à l'aide d'assertions afin de vérifier dynamiquement ses invariants (sur l'*aliasing* en particulier).

9. Remerciements

Je tiens à remercier Benjamin Monate et Virgile Prevosto pour leur relecture attentive d'une version préliminaire de cet article ainsi que Pascal Cuoq pour ses remarques avisées ayant notamment permis l'amélioration de la gestion du *hashconsing* par la bibliothèque. Julien Peeters a également contribué à l'implémentation de la sérialisation et de la désérialisation. Je remercie également les référés anonymes pour leurs commentaires judicieux.

Références

[1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI C Specification Language*, Octobre 2008.

[2] John Billings, Peter Sewell, and Mark Shinwell. Type-safe distributed programming for ocaml. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, Septembre 2006.

[3] Sylvain Conchon and Jean-Christophe Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, Septembre 2006.

[4] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Le foncteur sonne toujours deux fois. In *Journées Francophones des Langages Applicatifs*, Mars 2005.

[5] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. In *Trends in Functional Programming*, Avril 2007.

[6] Guy Cousineau. Tilings as a programming exercise. *Theoretical Computer Science*, 281(1-2) :207–217, 2002.

13. Bogues #4288 (*including a functor application containing side effect*) et #4550 (*functor application and value restriction*) du gestionnaire public de bogues d'*OCaml*.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, Californie, États-Unis, 1977. ACM Press.

[8] Pascal Cuoq and Damien Doligez. Hashconsing in an Incrementally Garbage-Collected System, A Story of Weak Pointers and Hashconsing in OCaml 3.10.2. In *ACM SIGPLAN Workshop on ML*, Victoria, British Columbia, Canada, Septembre 2008.

[9] Pascal Cuoq and Virgile Prevosto. *Documentation of Frama-C's value analysis plug-in*, Octobre 2008.

[10] The Frama-C development team. *Frama-C : Framework for Modular Analyses of C*. `http://frama-c.cea.fr`.

[11] The Eclipse Foundation. *Eclipse*. `http://www.eclipse.org`.

[12] A. P. Ershov. On programming of arithmetic operations. *Communication of the ACM*, 1(8) :3–6, 1958.

[13] Eiichi Goto. Monocopy and Associative Algorithms in Extended Lisp. Technical Report TR-74-03, University of Toyko, 1974.

[14] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st Symposium on Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[15] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd Symposium on Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[16] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5) :667–698, 1996.

[17] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3) :269–303, 2000.

[18] Xavier Leroy, (Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon). *The Objective Caml System*. `http://caml.inria.fr`.

[19] Donald Michie. Memo functions : a language feature with "rote-learning" properties. Research Memorandum MIP-R-29, Department of Machine Intelligence & Perception, Edinburgh, 1967.

[20] Donald Michie. Memo functions and machine learning. *Nature*, 218 :19–22, 1968.

[21] Julien Peeters. Participation à un outil de vérification de programmes. Rapport de stage DTSI/SOL/LSL/2008-08203/JP, CEA LIST, Saclay, Septembre 2008.

[22] Norman Ramsey. ML Module Mania : A Type-Safe Separately Compiled, Extensible Interpreter. In *ACM SIGPLAN Workshop on ML*, 2005.

[23] Julien Signoles. Calcul statique des applications de modules paramétrés. In *Journées Francophones des Langages Applicatifs*, pages 21–36, Janvier 2003.

[24] Julien Signoles. Une approche fonctionnelle du modèle vue-contrôleur. In *Journées Francophones des Langages Applicatifs*, Mars 2005.

[25] Julien Signoles and Virgile Prevosto. *Frama-C Plug-in Development Guide*, Octobre 2008.

[26] Jennifer Vesperman. *Essential CVS*. O'Reilly, 2nd edition, Novembre 2006.

[27] Mark Weiser. *Program slices : formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, 1979.

[28] Mark Weiser. Program slicing. In *5th International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

A. Preuve de la propriété de cloisonnement des projets

Dans cette annexe, nous démontrons la propriété 1 garantissant le cloisonnement entre projets. Cette propriété est rappelée ci-dessous.

Propriété 1 (Cloisonnement des projets) *Si les hypothèses 1, 2 et 3 sont vérifiées par chaque argument appliqué au foncteur* `Register`*, alors aucune version d'un état enregistré en appliquant ce foncteur n'est partagée avec une autre version d'un état d'un autre projet, ce qui peut être formellement exprimé par :*

$$\forall \text{ projets } p_1, p_2 \text{ tels que } p_1 \texttt{ <> } p_2, \forall \text{ états } s_1, s_2, \texttt{(find}_{s_1}\ p_1\texttt{).state != (find}_{s_2}\ p_2\texttt{).state}.$$

Pour rappel, les trois hypothèses mentionnées dans la propriété sont les suivantes.

$$\texttt{create ()} \text{ retourne une valeur fraîche} \tag{1}$$

$$\forall \text{ valeur } v, \texttt{copy } v \text{ retourne une valeur fraîche} \tag{2}$$

$$\forall \text{ valeurs } v_1, v_2 \text{ telles que } v_1 \texttt{ != } v_2, \texttt{set } v_1 \texttt{ ; get () != } v_2 \tag{3}$$

Pour mener à bien cette preuve, nous montrons en même temps que celle-ci la propriété suivante indiquant que la version locale d'un état est également cloisonnée, même si elle peut partagée le même espace que le projet courant.

Propriété 2 (Cloisonnement local) *Si les hypothèses 1, 2 et 3 sont vérifiées par chaque argument appliqué au foncteur* `Register`*, alors la version locale d'un état n'est pas partagée avec un état d'un projet différent du projet courant, ce qui peut être formellement exprimé par :*

$$\forall \text{ projet } p \text{ tel que } \texttt{is_current } p \texttt{ = false}, \forall \text{ états } s_1, s_2, \texttt{(find}_{s_1}\ p\texttt{).state != get}_{s_2}\ \texttt{()}.$$

Nous allons montrer que les propriétés 1 et 2 sont préservées par chaque appel aux fonctions `create`, `set_current` et `copy` en prouvant que chaque instruction du corps de ces fonctions les préservent. Pour chacune de ces trois fonctions, nous rappelons leur code. Afin d'améliorer la lisibilité de la preuve, nous numérotons les instructions du corps de la fonction, nous bêta-réduisons les appels de fonctions dont le corps est connu, tandis que les valeurs du corps du foncteur `Register` sont suffixées par un état s. Chaque ligne de code utilisant au moins une telle valeur porte en réalité, dans le code original, sur chacun des états enregistrés.

– **Cas de la fonction** `create`

```
let create =
  let cpt = ref 0 in
  fun () →
1   if cpt = -1 then invalid_arg "cannot create project: too many projects";
2   incr cpt;
    let p = !cpt in
3   Q.add p projects;
4   Hashtbl.add tbl_s p { state = create_s () };
5   if is_current p then set_s (find_s p).state;
    p
```

De manière immédiate, les instructions *1* à *3* préservent les deux propriétés, tandis que leur préservation par l'instruction *4* est garantie par l'hypothèse 1. Reste donc le cas de l'instruction *5*. Si p n'est pas le projet courant, ce qui est le cas s'il existait au moins un projet avant l'appel de

`create`, alors les propriétés sont trivialement préservées. Sinon (p est le projet courant), pour tout couple (p', s'), la valeur `(find`$_{s'}$ `p').state` n'a pas pu être modifiée par l'appel à `set`$_s$ car elle n'est pas visible hors du foncteur `Register` (rappelons que le type définissant le champ `state` est local au foncteur `Register`). La propriété 1 est donc préservée. En outre, l'hypothèse 3 implique que toute valeur différente de `(find`$_s$ `p).state` est physiquement différente de `get`$_s$ `()` après l'instruction *5*. La question ne se pose pas pour `(find`$_s$ `p).state` elle-même car `is_current p = true`. La propriété 2 est donc aussi préservée. Toutes les instructions du corps de la fonction `create` préservant les deux hypothèses, un appel à cette fonction les préserve donc.

– **Cas de la fonction** `set_current`

```
let set_current p =
1 if not (Q.mem p projects) then invalid_arg "set_current";
2 if is_current p then (find_s p).state ← get_s ();
3 Q.move_at_top p projects;
4 if is_current p then set_s (find_s p).state
```

De manière immédiate, les instructions *1* et *3* préservent les deux propriétés. Intéressons nous maintenant à l'instruction *2*. Si p n'est pas le projet courant, elle préserve trivialement les deux propriétés. Sinon, la seule valeur modifiée est `(find`$_s$ `p).state` qui devient physiquement égale à `get`$_s$ `()`. La propriété 2 est donc préservée car `is_current p = true`. À l'issue de l'instruction *2*, la valeur de `get`$_s$ `()` est ainsi physiquement égale à la valeur de l'état s du projet p et est différente des valeurs de tout état de tout autre projet (notons que p est l'unique projet courant). La propriété 1 est donc aussi préservée. Il reste maintenant le cas de l'instruction *4*. La preuve de la préservation des deux propriétés est similaire à celle de l'instruction *5* de la fonction `create`. Toutes les instructions du corps de la fonction `set_current` préservant les deux hypothèses, un appel à cette fonction les préserve donc.

– **Cas de la fonction** `copy`

```
let copy ?(src=current()) dst =
1 if is_current src then (find_s src).state ← get_s ();
2 (let v = find_s src in (find_s dst).state ← { v with state = copy_s v.state });
3 if is_current dst then set_s (find_s dst).state
```

L'instruction *2* correspond à l'instruction `States.copy src dst` dont le code précis a été omis jusqu'alors dans l'article. La preuve de la préservation des deux propriétés pour l'instruction *1* (respectivement *3*) est similaire à celle de l'instruction *2* (respectivement *4*) de la fonction `set_current`. Quant à l'instruction *2*, l'hypothèse 2 permet de garantir qu'elle préserve les propriétés visées. Toutes les instructions du corps de la fonction `copy` préservant les deux hypothèses, un appel à cette fonction les préserve donc.

Vers une programmation fonctionnelle en appel par valeur sur systèmes multi-coeurs : évaluation asynchrone et ramasse-miettes parallèle.

Luca Saiu

LIPN - UMR 7030
Université Paris 13 - CNRS
99, av. J.B. Clément - F-93430 Villetaneuse

Résumé

Les machines multi-coeurs modernes sont conçues pour exécuter de manière efficace des programmes assembleur *explicitement* parallèles. En effet, la technologie actuelle avance toujours moins dans la direction du parallélisme au niveau des instructions, qui ne demande pas une modification des programmes, et toujours plus vers le parallélisme au niveau des tâches qui, au contraire, demande une modification des programmes. Dans ce contexte, l'exploitation du parallélisme nécessite une modification soit des programmes assembleurs, en aval de la chaîne de compilation, soit des programmes d'un langage de plus haut niveau en amont de la chaîne. Autrement dit, l'exploitation des machines multi-coeurs peut suivre deux directions alternatives : changer le modèle de programmation ou bien cacher à l'utilisateur la complexité d'une telle tâche en déléguant au compilateur la parallélisation d'un programme de plus haut niveau, dans un style préférablement déclaratif. Dans les deux cas, les versions compilées doivent être efficaces sur ce type d'architecture. L'efficacité du support à temps d'exécution est donc cruciale. En particulier, lorsque le langage est de haut niveau et prévoit une gestion automatique de la mémoire, le ramasse-miettes est un goulet d'étranglement potentiel qui risque de limiter le gain de performance de l'application toute entière. Cet article est un retour d'expérience sur l'*association* de deux outils que nous avons implantés, un interpréteur d'un langage fonctionnel augmenté par une construction pour le calcul asynchrone et un ramasse-miettes pour architectures parallèles. La combinaison des deux outils offre à un utilisateur, humain ou compilateur, une manière d'exploiter le parallélisme matériel d'une machine multiprocesseur à mémoire partagée par un style de programmation déclaratif et essentiellement fonctionnel.

1. Introduction

La pratique de la programmation fonctionnelle montre à quel point l'adoption du modèle applicatif permet la réalisation de logiciels complexes de façon plus économique, particulièrement dans le cadre des langages typés statiquement. Par ailleurs, les performances se sont améliorées dans le temps et il est courant d'observer des compilateurs de langages fonctionnels qui génèrent du code ayant des performances comparables à celles d'un code impératif. Le compilateur de OCaml et certains compilateurs du langage SML en sont de bons exemples.

Pendant plusieurs années, les constructeurs de matériel ont focalisé leur offre sur des machines mono-processeur où le processeur devenait de plus en plus performant et complexe, mais d'une complexité dont l'utilisateur pouvait faire abstraction. En effet, l'apparition d'une nouvelle génération de processeurs induisait un accroissement *gratuit* des performances : tout processeur rendait plus efficace le code *existant* par une exploitation accrue du *parallélisme au niveau des instructions*, de la taille des

mémoires cache mais, surtout, par une augmentation souvent spectaculaire de la fréquence d'horloge.

On a mangé notre pain blanc (*the free lunch is over*, [21]) : si la loi de Moore reste valide [6] depuis 1975, l'augmentation du nombre de transistors par puce ne correspond plus à une augmentation de la fréquence d'horloge. Les producteurs de processeurs génériques se sont ainsi réorientés vers une architecture *multi-coeurs* à mémoire partagée (Symmetric Multi-Processing ou SMP) permettant l'avancement parallèle de plusieurs fils d'exécution ou *threads*. La complexité interne des processeurs [1], qui exploite le parallélisme au niveau des instructions, doit alors se réduire : pour inclure le plus grand nombre de coeurs dans une puce, chaque coeur doit occuper le moins de place possible quitte à renoncer aux structures de pipelining longues et quitte à réduire le degré (nombre de voies) de l'architecture superscalaire. Il ne s'agit pas d'un vrai choix de la part des producteurs : c'est une nécessité due à des limites physiques, reconnue de façon presque unanime [2], qui ne sera pas remise en question à moins d'une révolution technologique imprévue. Actuellement, des processeurs avec des centaines voire *milliers* de coeurs sont ainsi annoncés en cours de développement [4]. Or, programmer une telle puissance de calcul est loin d'être simple et l'opposition de Knuth à la direction prise par les constructeurs est recevable au moins dans ce sens. Pour approcher les limites théoriques des performances des processeurs multi-coeurs un modèle de programmation différent s'impose, tout au moins au niveau assembleur. Dans l'idéal, plusieurs fils d'exécution devraient pouvoir avancer sur les différents coeurs du système avec une distribution de la charge uniforme et sans demander excessivement de synchronisations. On passe de la forme de parallélisme au niveau des instructions à celle *au niveau des tâches*, une forme qui ne peut pas être «déduite» automatiquement par le processeur mais qui, au contraire, demande que le code soit *explicitement* parallèle, c'est-à-dire découpé en «tâches», threads ou processus. Ce découpage n'est pas une pratique nouvelle : une utilisation explicite de la concurrence est parfois pertinente dans la construction d'interfaces graphiques ou logiciels interactifs ; mais, il s'agit d'un besoin de modularité du programme, d'expressivité, plutôt qu'un souhait de parallélisation afin d'améliorer les performances. Toutefois, quelque soit l'objectif, les développeurs qui pratiquent la programmation concurrente s'exposent aux risques d'un modèle intrinsèquement complexe où le partage de données *mutables*, c'est-à-dire non persistantes, et les synchronisations explicites sont monnaie courante. Cette forme de contrôle explicite reste donc une technique de bas niveau essentiellement *impérative*, qui contraste avec le style fonctionnel et qu'il serait souhaitable de cacher derrière un style de programmation de plus haut niveau.

Les techniques de programmation parallèle le plus couramment utilisées ne sont pas de haut niveau, que le modèle de parallélisme soit à squelettes algorithmiques (*skeleton programming*), à flot de données (*dataflow*), ou data-parallèle imbriqué (*nested data parallel*), et que le paradigme de programmation soit au départ impératif, fonctionnel, logique ou autre. Dans le cadre impératif, le système OpenMP [19] mérite d'être cité comme une tentative plutôt réussie d'extension du C et de Fortran avec des constructions de haut niveau pour le parallélisme explicite avec partage de mémoire.

Plusieurs tentatives existent aussi dans le cadre fonctionnel quoique les tentatives se soient soldées par des échecs en parallélisme implicite pur. Avec moins d'ambition mais, sans doute, plus de réalisme à court terme, on peut imaginer l'introduction d'annotations ou de constructions syntaxiques spécifiques à la parallélisation de certaines parties du programme, ce qui est notamment le cas avec les constructions `future` ou `let-par` [5, 15, 7, 14, 18]. Ainsi, le langage JoCaml [8] est une extension de OCaml inspiré par le Join Calculus, un modèle de la concurrence à échange de messages proche du π-calcul. Ce système, tout comme le cadre MPI, correspond à un modèle de processus à *environnement local*. À contrario, notre étude repose sur un modèle à mémoire partagée, c'est-à-dire un modèle de concurrence à *environnement global*.

Dans cette multitude de possibilités, le paradigme fonctionnel semble constituer, a priori, une

1. Dans ce contexte, nous ignorerons la distinction entre coeur et CPU
2. Il existe sur ce choix l'opinion divergente de Donald Knuth [16], qui aurait insisté dans la direction monoprocesseur pour les difficultés évidentes de la programmation de telles machines.

meilleure base pour l'analyse automatique par la simplicité de sa sémantique et de son modèle d'exécution.

Dans un premier temps, nous présenterons le langage fonctionnel nanolisp. nanolisp intègre, en sus des constructions fonctionnelles élémentaires, la primitive `future` de Baker et Hewitt [14] permettant l'évaluation asynchrone d'une expression. nanolisp pourra être considéré de deux manières différentes : comme exemple de langage fonctionnel avec parallélisme explicite ou comme une étape intermédiaire ou finale de la traduction d'un langage fonctionnel implicitement parallèle.
Que nanolisp soit le point de départ ou bien un langage intermédiaire dans une hypothétique chaîne de compilation, ses performances dépendront de la présence d'un système de gestion de la mémoire capable de bien exploiter le parallélisme matériel. En effet, d'une part, il serait difficile d'imaginer un langage parallèle sans un support à temps d'exécution lui aussi parallèle. D'autre part, il serait difficile d'imaginer un langage fonctionnel sans une gestion automatique de la mémoire (allocation et récupération). La deuxième partie de ce travail décrira l'implantation d'un ramasse-miettes, appelé *epsilongc*, ses conditions de fonctionnement optimales. Ses performances sont comparées au ramasse-miettes de Bohem, seule alternative adaptable aujourd'hui aux systèmes multiprocesseurs à mémoire partagée. Pour nos mesures de performances, nous utiliserons nanolisp de façon assez particulière, comme prototype d'un support à temps d'exécution d'un langage fonctionnel parallèle.

2. nanolisp

nanolisp est un langage d'ordre supérieur à portée statique[3] de la famille Lisp, donc typé dynamiquement et basé sur les s-expressions. D'un point de vue utilisateur, il reprend un sous-ensemble de Scheme en l'étendant avec la construction `future`. L'interpréteur peut fonctionner en modalité interactive ou lire son entrée depuis un fichier.

Comme les autres dialectes Lisp, nanolisp est *homoiconique* : toutes les expressions valides du langage sont aussi des structures de données valides et peuvent être ainsi manipulées comme toute autre donnée. Cependant, *dans un souci de présentation*, nous décrirons d'abord une syntaxe utilisateur permettant de distinguer clairement les constructions importantes du langage.

De cette syntaxe utilisateur, seule la construction `future` présente des spécificités que nous détaillerons dans la sous-section suivante. En section 2.3 nous présentons succinctement une sémantique opérationnelle.

2.1. Syntaxe utilisateur

$e \rightarrow k \mid x \mid$ (**if** e e e) $\mid$ (**lambda** (x ... x) e) $\mid$ (**let** ((x e) ... (x e)) e) $\mid$ (e e ... e)
$\mid$ *cons*
$\mid$ (**quote** e)
$\mid$ (**future** e)

$k \rightarrow$ () $\mid$ `#t` $\mid$ `#f` $\mid$ *int*
cons $\rightarrow$ (e . e)
form $\rightarrow$ (**define** x e) $\mid$ e
prog $\rightarrow$ *form* $\mid$ *form prog*

Une expression e peut être, dans l'ordre des productions, une constante (liste vide ou *nil*, les booléens vrai et faux, un entier), une variable, une conditionnelle, une abstraction, un bloc, une application. Les trois dernières productions sont le *cons*, qui permet la construction de n-uplets ou listes, le `quote` qui

3. la clotûre statique concerne seulement les définitions locales (`let`) et non les définitions globales du top-level (`define`).

permet de bloquer l'évaluation d'un sous-programme et le `future` qui permet l'évaluation asynchrone d'une expression. Un programme est une séquence de *form*, c'est-à-dire une séquence de définitions ou évaluations d'expressions.

Abréviations syntaxiques Dans la tradition Lisp, la partie gauche d'un *cons* s'appelle *car* et la partie droite *cdr*. Les listes sont représentées de façon conventionnelle par des *cons* imbriqués à droite avec l'élément *nil* en dernière position. Par exemple, la liste des trois éléments `1`, `#f` et `toto` se représente par `(1 . (#f. (toto . ())))`. En raison de leur utilisation fréquente, une notation alternative permet de les représenter de façon plus pratique : si le cdr d'un *cons* est lui aussi un *cons* ou *nil*, alors il est possible d'omettre le point et les parenthèses autour du cdr. Autrement dit, le *cons* associe à droite. Par exemple la liste avec un seul élément `(a . ())` peut s'écrire `(a)`, la liste de trois éléments `(a . (b . c))` peut s'écrire `(a b . c)` et la liste `(1 . ((#t. 3) . (toto . ())))` peut s'écrire `(1 (#t. 3) toto)`.

La notation des listes introduit une ambiguïté *dans notre présentation* du langage lorsque un mot clef tel que `if`, `lambda`, `let`, `quote`, `future`, ou `define` paraît en première position après une parenthèse ouvrante. Dans la tradition Lisp, le langage est présenté avec les s-expressions qui évitent toute ambiguïté, certes, mais qui écrasent toutes les constructions syntaxiques au seul *cons*. L'ambiguïté sera donc levée au niveau sémantique en interprétant les expressions *e* du langage utilisateur comme s-expressions et en donnant une sémantique à ces dernières.

D'autres abréviations syntaxiques de Scheme sont supportées, entre autres celle facilitant le *quoting* : `(quote` *e*`)` pourra s'écrire `'`*e*.

Primitives Le langage contient les primitives habituelles sur les entiers (`+`, `-`, `*`,...), sur les n-uplets ou listes (`cons`, `car`, `cdr`,...), l'égalité structurelle et l'égalité physique (`equal?`, `eq?`). La syntaxe de l'application des fonctions primitives est identique à celle des fonctions ordinaires, elle est donc préfixe, jamais infixe : l'addition de deux entiers s'écrira `(+ 2 3)`.

2.2. Future

La construction `future` offre la possibilité d'un calcul asynchrone parallèle. Le paramètre de `(future` *e*`)` est une expression arbitraire. Son évaluation produit un nouvel objet de type *future* qui représente une calcul *asynchrone* par rapport à l'appelant. À chaque instant, cet objet peut se trouver dans deux états possibles que nous appellerons *achevé* et *inachevé*. L'objet démarre dans l'état inachevé et passe définitivement dans l'état achevé lorsque le calcul (potentiellement parallèle) de son *résultat* se termine. Un `future` peut être utilisé dans toute expression comme n'importe quelle autre valeur, même, et l'intérêt est justement là, dans l'état inachevé. D'un point de vue utilisateur un objet future peut être utilisé à la place de son résultat comme s'il était d'emblée achevé. En effet, les primitives du langage (`lambda`, `let`, `if`, ...) et les fonctions primitives (`+`, `cons`, `car`, ...) sont réalisées de façon à attendre l'accomplissement des objets future qu'elles prennent en paramètre seulement lorsque cela est *strictement nécessaire*. Si l'analogie avec les langages à évaluation paresseuse est naturelle, toutefois il convient d'avoir à l'esprit que l'évaluation d'un `future` est *anticipée* le plus possible et non retardée comme en sémantique paresseuse. Dans l'exemple suivant la fonction d'ordre supérieur *apply* est appelée avec en arguments la fonction identité et un `future` représentant le calcul asynchrone de l'addition des entiers 1 et 2.

```
(let ((apply (lambda (f x)
               (f x))))
  (apply (lambda (y) y)
```

```
        (future (+ 1 2))))
```

On voit dans cet exemple que l'utilisation d'un `future` en argument d'une fonction n'implique pas forcément l'attente de son accomplissement. En effet, la valeur générée par `(future (+ 1 2))` traversera l'application de `apply` et l'application de la fonction identité `(lambda (y) y)` sans aucune raison que ces applications attendent. Même si son évaluation peut être différée, elle aura lieu dans l'environnement de création, en s'alignant ainsi à une politique de portée statique.

Une fonction primitive additionnelle, `touch`, permet de forcer l'attente de l'accomplissement du calcul d'un objet future.

2.2.1. Usage de `future`

L'utilisation naturelle de la construction `future` est d'≪annoter≫ certaines expressions afin que l'exécution de celles-ci soit déclenchée par le système d'exploitation si tant est que des coeurs soient libres. Par exemple, en supposant d'avoir à disposition des fonctions d'inversion et d'addition de matrices, le calcul de $A^{-1} + B^{-1}$ où A et B sont des matrices, s'écrira ainsi :

```
(let ((inverted-a (matrix-invert a))
      (inverted-b (matrix-invert b)))
  (matrix-+ inverted-a inverted-b))
```

Or, le calcul des deux inverses pouvant être effectué en parallèle, nous pourrions encapsuler les appels d'inversion dans des [4] expressions `future` :

```
(let ((inverted-a (future (matrix-invert a)))
      (inverted-b (future (matrix-invert b))))
  (matrix-+ inverted-a inverted-b))
```

L'addition attendra ainsi la fin des deux inversions de matrice. Cet exemple montre une façon d'utiliser les `future` pour émuler la construction `let-par` de certains langages. Autrement dit, avec `future`, une simple extension de syntaxe (ou une macro dans le cas Lisp), permettra gratuitement la construction `let-par`.

Si l'utilisation de `future` est plutôt intuitive dans le cas de calculs non récursifs, elle devient moins évidente dans un cadre récursif. En effet, les objets future ont, bien sûr, un coût principalement dû aux synchronisations qu'ils déclenchent. On ne peut donc imaginer annoter systématiquement toute expression. La situation est beaucoup plus complexe lorsque le code dans un `future` contient un appel récursif, car il est souvent difficile de prévoir, a priori, combien d'objets future seront créés. Pour éviter de générer un grand nombre de calculs très courts, il est judicieux de ne rendre le calcul asynchrone qu'à partir d'une granularité minimale fixée. Dans des algorithmes du style diviser pour régner cela signifie utiliser les `future` jusqu'à un certain niveau maximal de profondeur des appels récursifs. Dans l'exemple suivant, la fonction de Fibonacci est calculée de façon asynchrone pour les arguments jusqu'au seuil `maximum-parameter-for-sequential-computation`, et de façon séquentielle pour les nombres inférieurs à ce seuil :

```
(define (sequential-fibonacci n)
  (if (< n 2)
      1
      (+ (sequential-fibonacci (- n 2))
```

4. Un seul serait déjà suffisant pour exécuter une des deux inversions dans un thread distinct et obtenir le même gain de performances.

```
        (sequential-fibonacci (- n 1)))))

(define (parallel-fibonacci n)
  (if (< n maximum-parameter-for-sequential-computation)
      (sequential-fibonacci n)
      (let ((fibonacci-of-n-minus-2 (future (parallel-fibonacci (- n 2))))
            (fibonacci-of-n-minus-1 (future (parallel-fibonacci (- n 1)))))
        (+ fibonacci-of-n-minus-2 fibonacci-of-n-minus-1))))
```

Dans un environnement de développement complet, on peut envisager que ce type de transformation (qui se traduit par une duplication du code) ne soit pas à la charge du développeur mais soit automatique ou semi-automatique (l'utilisateur indiquerait les seuils et une mesure pour calculer dynamiquement la granularité des sous-problèmes).

2.3. Sémantique operationnelle à petits pas

Pour décrire le fonctionnement du langage d'une façon fidèle à son implantation nous utiliserons une sémantique operationnelle à petits pas.

S-expressions. Une s-expression est définie inductivement comme un *atome* ou un *cons*, c'est-à-dire un couple de s-expressions.

$e \rightarrow atom \mid cons$
$atom \rightarrow$ `()` $\mid$ `#t` $\mid$ `#f` $\mid int \mid x$
$cons \rightarrow$ `(`e `.` e`)`

Les mêmes abréviations syntaxiques présentées dans la section 2.1 s'appliquent aux s-expressions. Ainsi, cette syntaxe permet d'encoder naturellement et sans plus d'ambiguïté le langage utilisateur (expressions, formes et programmes) présenté auparavant. Par exemple l'expression de l'utilisateur `(if (< 2 3) 0 x)` sera représenté par la liste dont le premier élément est le symbole `if` suivi des éléments `(< 2 3)`, `0` et `x`. Par la suite, nous utiliserons la lettre e pour les s-expressions.

Domaines. Le `quote` permet en Lisp de traiter les programmes comme données et représente sans doute un trait fondamental du langage. Il est particulièrement utile dans les calculs symboliques ; il peut contribuer à la génération dynamique d'une s-expression qui sera évaluée en tant que code avec la primitive [5] `eval`. Toutefois, `quote` complique la sémantique car il oblige à distinguer explicitement ce qui est réductible, ou *redex*, de ce qui est *valeur* non réductible. Cela nous amène à utiliser deux couleurs : le *vert* pour les redex et le *rouge* pour les autres, et aussi deux domaines distincts : les s-expressions $\mathbb{E}$ et les s-expressions colorées (à tous les niveaux de profondeur) $\mathbb{E}_c$. La syntaxe suivante définit les s-expressions colorées :

$e_c \rightarrow atom\ color \mid cons_c\ color$
$cons_c \rightarrow$ `(`e_c `.` e_c`)`
$color \rightarrow$ **red** $\mid$ **green**

Pour définir la sémantique operationnelle du langage nous utiliserons des atomes supplémentaires tels que les clôtures, les primitives et les objets futures. Cette extension concerne les deux langages $\mathbb{E}$ et $\mathbb{E}_c$:

$$atom \rightarrow \mathcal{C}(\rho, (x_1 ... x_n), e) \mid \mathcal{P}(p) \mid \mathcal{F}(t)$$

où :

5. `eval` est une primitive seulement pour une raison d'efficacité, mais est définissable dans le langage.

- $\rho \in Env$ est un environnement, $Env \triangleq \mathbb{S} \to \mathbb{E}_c$
- $\sigma \in TEnv$ est un environnement de tâches, $TEnv \triangleq \mathbb{T} \to (Env \times \mathbb{E}_c)$
- $p \in \mathbb{P}$ est une primitive, $\mathbb{P} \triangleq \mathbb{E}^* \to \mathbb{E}$
- $t \in \mathbb{T}$ est un identificateur de *tâche*
- $\mathbb{S}$ est l'ensemble des symboles, c'est-à-dire le langage du non-terminal x

2.3.1. Conventions méta-syntaxiques sur les couleurs.

Pour alléger la notation nous utiliserons du texte coloré pour représenter concisément la couleur associée à la racine du terme. Les termes seront aussi sur-lignés ou sous-lignés, selon la couleur, de façon à rendre ce document lisible en noir et blanc. Par exemple :

- $\overline{\mathtt{a}}$ signifiera `a red`
- $\underline{(\overline{\mathcal{P}(p)}\ .\ \underline{()})}$ signifiera `(`$\mathcal{P}(p)$ `red . () green) green`

Cette convention s'applique aussi aux méta-variables : la couleur de la méta-variable indique toujours la couleur de la *racine* du terme dénoté. L'abréviation des listes (cf. section 2.1) reste utilisable en supposant que la couleur de l'*épine dorsale* de la liste (tous les cons de sa structure et son nil) soit de la même couleur que la racine, c'est-à-dire du cons extérieur. Dans certaines règles la même méta-variable peut paraître avec deux couleurs différentes, par exemple :

$$\frac{\ldots}{\ldots\ \underline{e_1}\ \ldots\ \to\ \ldots\ \overline{e_1}\ \ldots}$$

Cela suppose une opération de re-coloriage en profondeur sur tout le terme (sans traverser les environnements) qui se définit de façon naturelle. Les opérations d'injection et d'oubli des couleurs sont aussi triviales. Le *bleu* avec un tilde sur le terme, comme par exemple dans $\widetilde{e}$, est utilisé comme *méta-couleur* pour indiquer une couleur quelconque.

2.3.2. Sémantique des expressions

Nous indiquerons par $\Gamma, \rho \ \vdash\ \sigma \rhd \widetilde{e}\ \to_e\ \sigma' \rhd \widetilde{e'}$ un pas de réduction d'une s-expression colorée dans l'environnement global $\Gamma \in Env$, l'environnement local $\rho \in Env$ et dans l'environnement des tâches $\sigma \in TEnv$. Les deux environnements ne sont pas modifiés par la transition des expressions. En revanche, l'environnement des tâches peut être modifié par la création et l'évolution des objets future.

Nous allons détailler les règles plus importantes ; la sémantique complète est fournie en annexe.

Atomes. Tous les atomes verts qui ne sont pas des symboles (variables) se disent dans le jargon Lisp *auto-évaluantes* car se réduisent à eux-mêmes en changent simplement de couleur :

$$[\text{Atome}]\ \frac{}{\Gamma, \rho\ \vdash\ \sigma \rhd \underline{a}\ \to_e\ \sigma \rhd \overline{a}}\ a \text{ est un atome et n'est pas un symbole}$$

Quote. Le quote permet d'éviter l'évaluation d'une s-expression en la traitant comme une donnée ; l'argument est renvoyé comme résultat en lui associant la couleur rouge :

$$[\text{Quote}]\ \frac{}{\Gamma, \rho\ \vdash\ \sigma \rhd \underline{(\underline{\mathtt{quote}}\ \underline{e})}\ \to_e\ \sigma \rhd \overline{e}}$$

Abstraction-application. Les règles correspondent au choix d'un langage en appel par valeur (arguments en rouge) et avec une clôture statique de l'environnement local. Dans la règle [Application$_1$] la contrainte sur $\widetilde{e_0}$ évite qu'une construction syntaxique du langage utilisateur soit interprétable comme application.

$$[\text{Abstraction}]\ \frac{}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\texttt{lambda}\ \underline{(\underline{x_1}\ \ldots\ \underline{x_n})}\ \underline{e})} \ \rightarrow_e\ \sigma \rhd \overline{\mathcal{C}(\rho, (x_1\ \ldots\ x_n), \underline{e})}}$$

$$[\text{Application}_1]\ \frac{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{e_i} \ \rightarrow_e\ \sigma' \rhd \widetilde{e'_i} \qquad \widetilde{e_0} \notin \{\underline{\texttt{if}}, \underline{\texttt{lambda}}, \underline{\texttt{let}}, \underline{\texttt{quote}}, \underline{\texttt{future}}\}}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\widetilde{e_0}\ \ldots\ \underline{e_i}\ \ldots\ \widetilde{e_n})} \ \rightarrow_e\ \sigma' \rhd \underline{(\widetilde{e_0}\ \ldots\ \widetilde{e'_i}\ \ldots\ \widetilde{e_n})}}\ 0 \leq i \leq n$$

$$[\text{Application}_2]\ \frac{\Gamma, \rho_1[x_1 \mapsto \overline{e'_1}, \ldots, x_n \mapsto \overline{e'_n}] \ \vdash\ \sigma \rhd \underline{e} \ \rightarrow_e\ \sigma' \rhd \widetilde{e'}}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\overline{\mathcal{C}(\rho_1, (x_1\ \ldots\ x_n), \underline{e})}\ \overline{e'_1}\ \ldots\ \overline{e'_n})} \ \rightarrow_e\ \sigma' \rhd \underline{(\overline{\mathcal{C}(\rho_1, (x_1\ \ldots\ x_n), \widetilde{e'})}\ \overline{e'_1}\ \ldots\ \overline{e'_n})}}$$

$$[\text{Application}_3]\ \frac{}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\overline{\mathcal{C}(\rho_1, (x_1\ \ldots\ x_n), \overline{e'})}\ \overline{e'_1}\ \ldots\ \overline{e'_n})} \ \rightarrow_e\ \sigma \rhd \overline{e'}}$$

Future La construction `(future` e`)` démarre un calcul asynchrone qui procédera en parallèle dans l'environnement des tâches.

$$[\text{Fork}]\ \frac{}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\texttt{future}\ \underline{e})} \ \rightarrow_e\ \sigma[t \mapsto (\rho, \underline{e})] \rhd \overline{\mathcal{F}(t)}}\ t \text{ fraîche}$$

$$[\text{Inachevé}]\ \frac{\Gamma, \rho_1 \ \vdash\ \sigma \rhd \underline{e_1} \ \rightarrow_e\ \sigma' \rhd \widetilde{e'_1}}{\Gamma, \rho \ \vdash\ \sigma[t \mapsto (\rho_1, \underline{e_1})] \rhd \widetilde{e} \ \rightarrow_e\ \sigma'[t \mapsto (\rho_1, \widetilde{e'_1})] \rhd \widetilde{e}}$$

$$[\text{Achevé}]\ \frac{}{\Gamma, \rho \ \vdash\ \sigma[t \mapsto (\rho_1, \overline{e'_1})] \rhd \mathrm{C}[\overline{\mathcal{F}(t)}] \ \rightarrow_e\ \sigma[t \mapsto (\rho_1, \overline{e'_1})] \rhd \mathrm{C}[\overline{e'_1}]}\ \text{pour tout contexte C[.]}$$

La règle [Fork] créé un identificateur de tâche t, le *ticket*, et un lien dans l'environnement des tâches σ entre le ticket et la clôture de l'expression à évaluer dans l'environnement local courant ρ. L'expression `(future` e`)` se réduit en $\overline{\mathcal{F}(t)}$ qui se comporte comme une valeur (non redex). Cependant, l'objet $\overline{\mathcal{F}(t)}$ n'est pas vraiment une valeur finale : il sera remplacé par le résultat du calcul lorsque la tâche sera achevée (règle contextuelle [Achevé]). La règle [Inachevé] est la vraie source de non-determinisme du système [6] : puisque aucune contrainte n'est imposée sur la forme de $\widetilde{e}$, la règle peut s'appliquer à tout moment dès lors qu'une s-expression est réductible dans l'environnement des tâches.

Primitive. L'objet $\overline{\mathcal{F}(t)}$ généré par la règle [Fork] peut être donné à (et renvoyé par) une fonction Lisp, pendant l'évaluation de la tâche t. En revanche, dans l'implantation actuelle toutes les primitives sont *eager*, c'est-à-dire que toutes se bloquent jusqu'au moment où tous leurs arguments sont achevés. Il existe actuellement une seule exception, la primitive `cons` qui n'attend pas la terminaison de ses arguments pour rendre son résultat (le *cons* de ses deux arguments).

$$[\text{Primitive}]\ \frac{p = \mathit{cons} \ \text{ ou } \ \overline{e_i} \neq \overline{\mathcal{F}(_)} \text{ pour tout } i}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\overline{\mathcal{P}(p)}\ \overline{e_1}\ \ldots\ \overline{e_n})} \ \rightarrow_e\ \sigma \rhd \overline{e}}\ p(e_1, \ldots, e_n) = e$$

Il est supporté qu'un objet future, une fois achevé, puisse se réduire à un autre objet future : dans le cas où p n'est pas *cons*, la règle [Primitive] ne demande pas seulement que les arguments soient terminés, mais demande qu'ils ne soient plus des objets future.

La primitive $\mathcal{P}(\mathit{touch})$ attend jusqu'au moment où son argument s'achève en une valeur non future et la renvoie. D'un point de vue mathématique il s'agit tout simplement de la fonction identité, mais son comportement est eager.

6. Au delà de la règle [Application$_1$] qui pourrait être rendue déterministe en spécifiant un ordre d'évaluation.

2.3.3. Sémantique des programmes et des form

La sémantique des programmes et des form est standard et donnée en annexe. Un programme constitué d'une séquence non vide de form f_0 ... f_n est évalué à partir de l'environnement global Γ_0 contenant les primitives et de l'environnement de tâches vide. La sémantique des programmes s'obtient par clôture transitive de la relation définie.

2.4. Implantation de nanolisp

Une version préliminaire du code est disponible pour les relecteurs à l'adresse `http://194.254.173.145/repos/code`. et sera publié prochainement comme logiciel libre.

Un interpréteur de nanolisp a été écrit en C selon le modèle eval/apply [1]. Les S-expressions sont implantées de façon traditionnelle (par exemple similaire à [13]), avec uniquement les valeurs *boxed* sur le tas et les valeurs *unboxed* avec les trois bits moins significatifs utilisés comme annotation de type (*tag*) à temps d'exécution.

Implantation de `future`. Le moteur d'exécution contient une file des objets future dont l'évaluation n'a pas encore été amorcée. Les objets future peuvent donc être dans 3 états, *achevé*, *inachevé* et l'état signifiant l'évaluation non encore amorcée, que nous appellerons *prévu* par la suite. Cet état apparaît dans l'implantation et n'est pas significatif pour l'utilisateur. Dans cet état, les objets future attendent qu'un thread se libère pour démarrer leur évaluation, de façon comparable à des processus qui, dans l'état prêt, attendent que le système d'exploitation leur donne la disponibilité d'une ressource de calcul. À chaque objet future est associé une structure de synchronisation (variable de condition) permettant aux threads de nanolisp d'attendre passivement, sans occuper un processeur, lorsque ils sont en attente de la fin du calcul d'un calcul asynchrone. Le résultat du calcul est enregistré dans un champ prévu à cet effet dans la structure de données représentant l'objet future. Un sémaphore global permet de compter le nombre de coeurs libres et permet d'éviter ainsi qu'un thread amorce l'exécution d'un objet future lorsque tous les coeurs sont déjà occupés par l'application. De cette manière, on ne résoudra jamais plus d'objets future à la fois que de coeurs disponibles. Un jeu de threads, appelés *worker threads*, réalise l'environnement des tâches de la sémantique. Chaque thread exécute en boucle les opérations suivantes :

- exécution de l'opération P sur le sémaphore des coeurs libres
- extraction d'un objet future de la file des prévus,
- évaluation par `eval` de l'expression associée dans son environnement de fermeture,
- mise à jour du champ contenant le résultat,
- exécution de l'opération V sur le sémaphore des coeurs libres
- réveil de tous les threads en attente sur la structure de synchronisation associée.

Représentation des environnements Les environnements sont réalisés avec des techniques standards et efficaces. Au cours de l'analyse syntaxique, chaque symbole est associé à une structure de données unique (*interning*). Dans cette structure de données se trouve la valeur du symbole dans l'environnement global. Les environnements locaux sont *plats* ou *chainés* selon la construction qui les génère : on alloue les segments d'environnement tant que possible sur la pile (`let`, application), sinon sur le tas (`lambda`, `future`).

3. *epsilongc*, un ramasse-miettes scalable

Sans un bon ramasse-miettes parallèle, nanolisp ne donnerait pas de résultats acceptables car une partie séquentielle, même mineure, dans un code majoritairement parallèle peut affecter négativement

les performances de toute l'application.

3.1. Scalabilité et efficacité

Les performances d'un programme se mesurent typiquement par le temps nécessaire à compléter le calcul. Dans le cadre des machines parallèles on peut aussi se poser la question de savoir si le programme exploite le parallélisme matériel et cela dans quelle mesure. Les notions d'*efficacité* et de *scalabilité* mettent en rapport le temps d'exécution réel et idéal. *Dans l'idéal*, en passant d'une machine mono-processeur à une machine avec k processeurs, on souhaiterait que le temps d'exécution total du programme se divise par k, c'est-à-dire se réduise de t_1 à $t_k = \frac{t_1}{k}$, ce qui signifierait avoir une exploitation des processeurs à 100%. La scalabilité s_k est le rapport entre les temps réels d'exécution t_1 et t_k, l'efficacité e_k est le rapport entre la scalabilité et le nombre de processeurs :

$$s_k \triangleq \frac{t_1}{t_k} \qquad\qquad e_k \triangleq \frac{s_k}{k}$$

Naturellement, pour tout k nous avons toujours $0 < e_k \leq 1$ et $0 < s_k \leq k$. Si, par exemple, un programme avait besoin de 80 secondes sur un mono-processeur et de 20 secondes sur un 8 coeurs, l'efficacité e_8 serait $\frac{1}{2}$ et la scalabilité s_8 serait 4 : intuitivement le parallélisme aurait été exploité à 50%, comme si on l'avait parfaitement exploité avec 4 processeurs. Pour résumer, on souhaite toujours une efficacité 1 et une scalabilité k. Malheureusement, l'efficacité diminue avec l'augmentation du nombre de processeurs à cause de l'augmentation des conflits d'accès aux ressources partagées (mémoire, bus, disques,..). Ce phénomène se manifeste à tous les niveaux : matériel, système d'exploitation et application. Le défi est celui de retarder le fléchissement de la courbe de scalabilité réelle et d'approcher tant que possible la *scalabilité idéale* k.

3.2. L'obligation de scalabilité

Dans un langage de haut niveau avec gestion automatique de la mémoire, le ramasse-miettes est partie intégrante du support à temps d'exécution du code exécuté. Cette partie peut affecter lourdement les performances du programme dans sa totalité, que le programme soit compilé ou interprété comme dans nanolisp. Pour mesurer les conséquences d'un éventuel ramasse-miettes séquentiel rattaché à un programme parallèle, il n'est pas inutile de rappeler la loi d'Amdahl [2] et ses conséquences. La loi d'Amdahl exprime la scalabilité [7] à laquelle on peut s'attendre en augmentant le degré de parallélisme d'une *partie* et non de la *totalité* du système.

Loi de Amdahl : Soit p le pourcentage du temps d'exécution d'un programme séquentiel pouvant être parallélisé avec une scalabilité s_p. Alors, le *scalabilité totale* du programme sera donné par la formule :

$$s_t = \frac{1}{(1-p) + \frac{p}{s_p}}$$

Ainsi, sous l'hypothèse vraisemblable que la gestion automatique de la mémoire coûte 10% de l'exécution séquentielle du programme, donc $p = 0.9$, et en supposant, cette fois de façon très optimiste, que tout le programme soit idéalement parallélisable, donc s_p égale au nombre de processeurs, nous obtiendrons : avec 4 processeurs $s_t \simeq 3.08$, avec 8 processeurs $s_t \simeq 4.71$, avec 16 processeurs $s_t \simeq 6.4$, avec 32 processeurs $s_t \simeq 7.8$, avec 64 processeurs $s_t \simeq 8.77$, avec 512 processeurs $s_t \simeq 9.83$.

En passant à la limite, pour $s_p \rightarrow \infty$ nous aurions $s_t \rightarrow \frac{1}{1-p} = 10$: c'est-à-dire que, malgré un nombre arbitrairement grand de processeurs, l'augmentation de la vitesse d'exécution du

7. Nous ignorons ici la nuance entre la notion de *speedup* évoquée par Amdahl et la scalabilité comme nous l'avons définie.

programme resterait inférieure à 10. Le ramasse-miettes d'un langage fonctionnel est donc une partie du système qui doit être impérativement parallélisée au risque, sinon, de devenir rapidement le goulet d'étranglement des applications.

3.3. La mémoire partagée

Pour que les versions compilées de nos programmes soient efficaces sur les machines actuelles, le système de gestion de la mémoire doit être programmé pour exploiter au mieux les spécificités de ces architectures. Nous allons donc passer en revue les traits qui nous paraissent essentiels. Il existe plusieurs architectures de mémoire qui permettent à des processeurs différents d'avoir accès aux mêmes modules de mémoire physiques. Le modèle principal aujourd'hui est SMP [8].

3.3.1. Le modèle de mémoire des machines SMP

Les machines SMP adoptent un modèle à mémoire partagée où le temps d'accès à une cellule de mémoire arbitraire est uniforme, indépendamment du coeur qui y accède. Depuis quelques années, la tendance des industriels, pour des raisons technologiques, est d'intégrer plusieurs *coeurs* dans une seule puce. Le modèle SMP conçu pour les anciennes machines avec un processeur par puce, reste toutefois dans la majorité des cas une très bonne approximation même avec plusieurs coeurs par puce.

Il existe plusieurs variantes de l'architecture SMP plus ou moins scalables par rapport au nombre de processeurs. Nous montrons deux exemples réalistes dans la Figure 1.

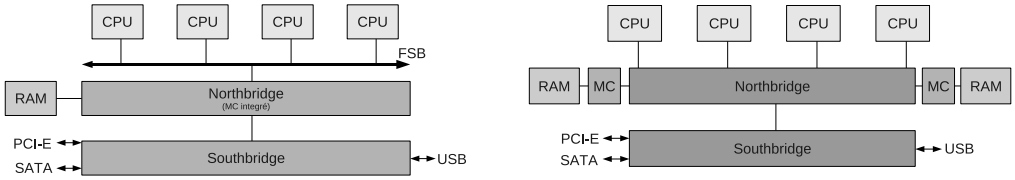


FIGURE 1 – Deux exemples d'architecture SMP. À gauche, l'accès à la mémoire se fait par le *Front Side Bus* (FSB) et par le contrôleur de mémoire intégré au Northbridge, partagés par l'ensemble des processeurs. Ces deux goulets d'étranglement sont éliminés dans l'architecture à droite, plus efficace et scalable. Les dessins sont une adaptation des originaux dans [11].

3.3.2. Localité et cache

Tous les processeurs modernes ont accès à la mémoire principale, située sur une autre puce, par l'intermédiaire d'une mémoire, appelée cache, de petite taille mais très rapide, installée sur la même puce. Typiquement, le cache est structuré à plusieurs niveaux, chacun contenant l'ensemble des *lignes* [9] les plus récemment utilisées, copiées à la demande du niveau de mémoire immédiatement supérieur dans la chaîne. La chaîne des mémoires cache qui amène du processeur jusqu'à la mémoire principale va de la plus petite et rapide à la plus grande et lente. L'existence du cache est transparente pour le développeur mais sa présence a des effets importants sur les performances des programmes. Pour qu'elles soient exploitées au mieux, les accès mémoire doivent être le plus possible *locaux*, c'est-à-dire que les cellules contiguës doivent être accédées le plus possible à des instants proches dans le temps.

8. Le modèle NUMA pourrait se développer à l'avenir mais demande un modèle de programmation très différent.
9. Une *ligne* est une séquence de mots de mémoire contiguës, de longueur dépendante de l'architecture.

3.4. Les hypothèses de la programmation fonctionnelle

Au niveau d'abstraction du système de gestion de la mémoire, un programme fonctionnel ou un interpréteur d'un langage fonctionnel comme nanolisp, perd les séduisantes propriétés mathématiques qu'il avait à un niveau d'abstraction supérieur. Au contraire, à ce niveau il se réduit à un programme impératif parallèle, mais avec des schémas d'allocation et d'accès assez particuliers. Les spécificités qu'on peut lui reconnaître sont :

- le taux d'allocation est élevé,
- la grande majorité des objets alloués a un nombre limité de *formes*, c'est-à-dire un nombre limité de combinaisons de *taille*, *alignement* et *tag*, où le tag est une valeur numérique représentant un constructeur d'un type algébrique [10],
- sous l'hypothèse d'appel par valeur, les affectations implicites [11] utilisées dans un langage paresseux ne sont plus nécessaires,

epsilongc est un ramasse-miettes générique et indépendant du langage mais conçu pour être particulièrement efficace lorsque toutes ces hypothèses sont réunies.

3.5. Formes, sources et pompes

Il est possible de représenter en mémoire de façon compacte une série d'objets de même forme en factorisant le tag de chaque objet : ce tag sera représenté une seule fois par espace, appelé *page* [12], dédié à l'enregistrement de la série d'objets. Cette technique est connue sous le nom de BIBOP [20, 17] depuis plusieurs décennies, mais se révèle particulièrement indiquée dans le cadre des architectures de mémoire modernes dont nous avons parlé plus haut, et encore mieux sous nos hypothèses. BIBOP est aussi une bonne opportunité pour paralléliser l'allocation car, dans cette phase, la page peut être réservée en exclusivité à un thread. Ce dernier pourra donc créer des nouveaux objets *sans besoin de synchronisation*, jusqu'au moment où la page sera pleine. C'est à ce moment que la page devra être remplacée par une page de même forme mais contenant de l'espace libre.
Étant donné une forme :

- chaque thread dispose d'une structure de données locale, non partagée, appelée *pompe*, lui permettant d'accéder à la page qui lui est actuellement réservée pour l'allocation des objets de la forme,
- une structure globale, appelée *source*, partagée par tous les threads, enregistre la liste des pages existantes pour les objets de la forme.

Le concept de pompe aide l'utilisateur dans l'usage du ramasse-miettes en permettant une abstraction simplificatrice. Comme illustré par la Figure 2, lorsque l'utilisateur aura besoin de créer un objet, il en demandera l'allocation à la pompe de forme adéquate. Il s'intéressera aux sources seulement au moment de l'initialisation, pour créer ses pompes, et pourra toujours ignorer les pages et la complexité de leur gestion.

Pour l'utilisateur, la pompe est une ressource inépuisable d'objets de la forme concernée. La récupération (*collection*) de ces objet est déclenchée automatiquement quand cela s'avère nécessaire. L'implantation actuelle de la récupération est basée sur un simple algorithme de marquage et nettoyage (*mark-sweep*) parallèle.

10. Dans un langage typé dynamiquement, comme un dialecte Lisp, le tag permettra de représenter le type des objets à temps d'exécution.
11. Un langage paresseux est habituellement implanté avec des fermetures qui sont destructivement remplacées avec les valeurs calculées peu à peu, ou avec des machines virtuelles à réduction de graphes réalisées dans un style impératif.
12. La notion de page n'est pas en relation avec celle homonyme du système d'exploitation.

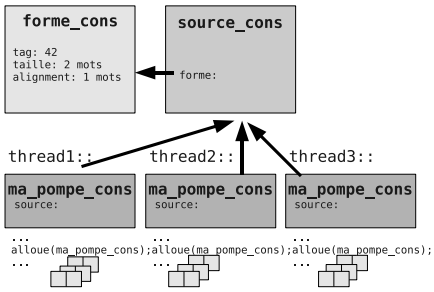


FIGURE 2 – Dans cet exemple, trois threads allouent en parallèle des objets de forme `cons`, en utilisant chacun sa propre pompe.

3.5.1. Notes sur l'implantation de *epsilongc*

Nous allons expliquer dans cette section comment les hypothèses de la section 3.4 et l'architecture SMP de machines actuelles ont conditionné nos choix d'implantation. L'idée principale du projet est de séparer en mémoire les données et les méta-données, en particulier le tag. Les accès aux tags ont tendance à être fréquents dans les langages fonctionnels et donc critiques pour les performances. Nous avons donc pris l'option de mémoriser les méta-données seulement *une fois par page* plutôt que *une fois par objet* boxed. Dans notre version de BIBOP on peut non seulement économiser de la mémoire mais on peut surtout faire une utilisation plus rationnelle de l'espace, très limité, de la mémoire cache, en réduisant les défauts de cache (*fault*) et par conséquent la pression sur la mémoire.

Les pages sont allouées avec un alignement égal à leur dimension qui, en octets, est une puissance de 2. Il est donc possible d'accéder à l'en-tête (*header*) d'une page, donc aux méta-données, par une simple operation de masquage entre bits. Cela ne coûte que 2 ou 3 instructions assembleur. Dans le contexte d'un langage fonctionnel cette solution, qui serait déjà avantageuse dans le cas séquentiel, devient particulièrement importante sur une machine SMP où la pression sur la mémoire est directement proportionnelle au nombre de processeurs.

La direction prise rend difficile un changement des méta-données en cours de route comme il serait nécessaire dans un langage paresseux où les clôtures sont sans cesse converties en objets de formes différentes. Pour cette raison, notre variante de BIBOP s'adapte mieux à un langage à évaluation immédiate (*eager*). Les objets future de nanolisp ont un caractère semblable aux valeurs paresseuses et, par conséquent, ne peuvent pas être très efficaces avec notre BIBOP : accéder à une valeur par un objet future achevé demande une redirection de pointeur supplémentaire.

Travaux similaires. Notre version de BIBOP est semblable à celle de E. Ulrich Kriegel dans [17]. En revanche, l'implantation parallèle de la libération de mémoire (*collection*) est semblable à [3]. D'autres implantations de BIBOP comme [12] et le ramasse-miettes de Boehm *ne mémorisent pas* les méta-données (dans notre acception) dans les en-têtes des pages. Il existe des travaux plus orientés à diminuer les *pauses*[13] qui sont beaucoup moins critiques pour les applications parallèles. Parmi ces travaux, se trouvent deux ramasse-miettes très sophistiqués qui, à la différence de *epsilongc*, sont concurrents et générationnels : celui de Leroy-Doligez [10], qui se base sur l'absence d'un opérateur d'égalité physique, et celui de Doligez-Gonthier [9]. Dans les deux cas, la libération de la mémoire sur la vielle génération (*tenured generation*) est séquentielle.

13. temps où le programme reste bloqué à cause du ramasse-miettes.

4. Étude expérimentale quantitative du ramasse-miettes avec nanolisp

nanolisp combiné à *epsilongc* a été testé sur un jeu de 5 programmes et comparé avec nanolisp combiné au ramasse-miettes de Boehm. Ces programmes ont été construits de façon à étudier le comportement quantitatif du système de gestion automatique de la mémoire. Par conséquent, ils ne pourraient pas être classés de façon très naturelle dans un schéma de calcul parallèle classique. Le parallélisme y est utilisé massivement, voire à l'excès : les *mêmes* opérations sont exécutées à l'identique, depuis différents threads, pour produire les mêmes résultats. Testé sur une machine à huit processeurs, le calcul est toujours exécuté huit fois et, selon le nombre de processeurs utilisés, un nombre différent de calculs avancera réellement en parallèle. Par exemple, en utilisant 4 processeurs, les huit répétitions seront exécutées quatre à la fois et le temps, dans l'idéal, sera le double du cas avec 8 processeurs disponibles.

Temps d'exécution total								
Nb de processeurs	1		2		4		8	
Ramasse-miettes	epsilon.	Boehm	epsilon.	Boehm	epsilon.	Boehm	epsilon.	Boehm
`allocate`	42.15	46.08	21.21	24.65	10.81	12.88	5.75	7.00
`kriegel`	81.37	82.45	40.64	43.69	20.45	21.66	10.4	11.15
`primes`	31.75	33.66	15.93	17.68	8.09	8.98	4.27	5.00
`primes-wide`	32.95	34.67	16.6	17.92	8.61	9.62	4.66	5.83
`sequential`	78.19	-	59.31	-	48.38	-	44.4	-

FIGURE 3 – Temps d'exécution en secondes de nanolisp avec *epsilongc* et avec le ramasse-miettes de Boehm, par programme et par nombre de processeurs utilisés. Chaque temps est le meilleur de trois exécutions.

Description des programmes utilisés. `allocate` alloue en boucle un nombre aléatoire d'éléments en les rangeant dans une liste ; le nombre total d'éléments par thread est donné en paramètre ; `kriegel` est une simple traduction en nanolisp de la fonction *reverse* sur les listes, intentionnellement inefficace, présentée dans [17] ; `primes` calcule la liste des nombres premiers inférieurs à un paramètre donné en utilisant des fonctions d'ordre supérieur sur les listes ; pour ce faire, il génère une quantité importante de blocs à recycler ; `primes-wide` est identique à `prime` mais exécuté sur une version modifiée de nanolisp où les `cons` sont représentés avec quatre mots de mémoire au lieu de deux ; `sequential` est un programme *séquentiel* qui construit une seule et grande structure arborescente ; il sert à mesurer la scalabilité du récupérateur.

Pour obliger *epsilongc* à récupérer plus souvent sur la machine de test [14] nous avons fixé la taille du tas à 512Mo. Pour le reste epsilongc a été utilisé dans sa configuration de défaut. Utilisé sur deux processeurs ou plus, le ramasse-miettes de Boehm montre une mauvaise scalabilité (cf. Figure 4). Ses performances augmentent en revanche de façon spectaculaire si on le configure en utilisant les variables d'environnement `GC_INITIAL_HEAP_SIZE` et `GC_MAXIMUM_HEAP_SIZE` : la meilleure valeur pour

14. Dell Precision T7400 avec deux quatre-coeurs Intel Xeon (EM64T) puces à 3GHz, 8Gb de RAM et un FSB à 1.3Ghz. L1I et L1D de 32K par coeur, associatives à 8 voies. L2s sont de 6Mb, associatives à 24 voies, une par *couple* de coeurs ; les lignes de cache sont de 64 octets. Le système d'exploitation est un GNU/Linux debian ≪unstable≫ avec un noyau Linux 2.6.26 et la GNU libc 2.7 ; les logiciels importants sont compilés à 64 bits avec GCC 4.3.2, en utilisant les options `-O3 -fomit-frame-pointer`. La version du ramasse-miettes de Boehm est la *7.1alpha3-080224*, configurée avec les options `--enable-threads=posix --disable-cplusplus --enable-parallel-mark --disable-gcj-support --disable-java-finalization`. Les tests ont été effectués en modalité single-user, sans autres tâches exécutées sur la machine.

notre machine de test semble être aux alentours de `5000000000` (5Go) pour les deux variables. Toutes les mesures du récupérateur de Boehm, exceptée la ligne «Boehm (default)» de la Figure 4, ont donc été réalisées avec les variables affectées à cette valeur optimale. Cette configuration a donné un certain avantage au ramasse-miettes de Boehm qui disposait ainsi de 5Go pour le tas contre les 0.5Go de *epsilongc*.

Les temps d'exécutions sont reproductibles avec des différences de quelques dizaines de millesecondes pour tous les programmes, lorsque la machine n'est pas chargée par d'autres tâches. La Figure 3 montre les meilleurs temps des trois exécutions.

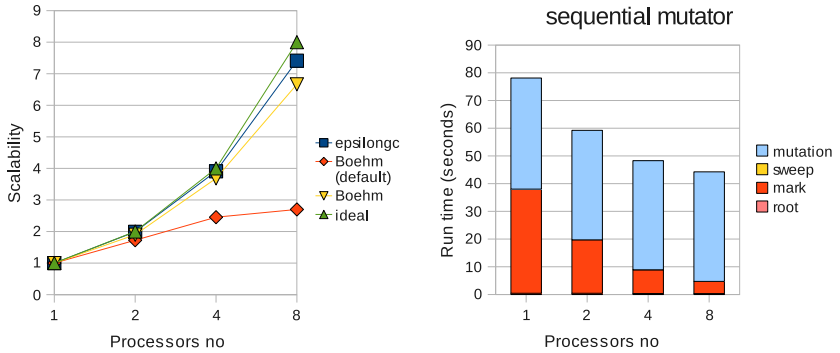


FIGURE 4 – Scalabilité moyenne de l'application et temps de récupération (ce dernier seulement avec epsilongc). Les valeurs de scalabilité à gauche n'incluent pas le test `sequential`.

4.1. Interprétation des résultats

Le système *epsilongc* et particulièrement son allocateur, qui prend le plus de temps processeur, se montre très scalable, avec une scalabilité moyenne de 7.41 sur 8 processeurs, contre une scalabilité de 6.66 avec le ramasse-miettes de Boehm. Sur 8 processeurs, le temps d'exécution avec *epsilongc* est de 20% en moyenne plus court que celui avec le système de Boehm. Le programme `sequential` montre que le marquage peut devenir le goulet d'étranglement de l'application quand la majorité du tas est occupée par des objets en utilisation. Le même exemple montre toutefois que le marquage approche la scalabilité idéale sur des structures de données non linéaires. Au delà de la scalabilité, les performances absolues sont aussi à l'avantage de *epsilongc* (cf. table de la Figure 3).

5. Conclusion

La plupart des machines récentes sont multi-coeurs, donc avec un parallélisme technologique spécifique, au niveau des tâches plutôt qu'au niveau des instructions. D'une part, les langages modernes doivent exploiter ce parallélisme à un niveau d'abstraction suffisamment élevé et, d'autre part, il faut que les versions compilées des programmes soient efficaces sur ces architectures. Avec l'objectif de ce type d'architecture, nous avons implanté un prototype de langage fonctionnel, nanolisp, incluant la construction `future`. Nous avons concentré nos efforts sur le ramasse-miettes, goulet d'étranglement classique pour la gestion automatique de la mémoire. Les résultats du couple nanolisp-*epsilongc* montrent de meilleures performances en termes de scalabilité par rapport au couple nanolisp-*Boehm*.

Les deux implantations, nanolisp et *epsilongc*, seront publiées prochainement comme logiciels libres. Il serait intéressant de tester *epsilongc* en substituant celui-ci aux ramasse-miettes à l'oeuvre dans des implantations de langages fonctionnels plus complètes comme OCaml ou GHC. Nous pourrions ainsi

tester le ramasse-miettes sur de “vrais” programmes et, surtout, vérifier si le gain obtenu par la parallélisation peut rattraper, et pour quelle architecture multi-coeurs, la remarquable implantation séquentielle du ramasse-miettes standard de OCaml.

Références

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 2nd edition, July 1996.

[2] Gene Amdahl. Validity of the single-processor approach to achieving large-scale computing requirements. *Computer Design*, 6(12) :39–40, 1967.

[3] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *PLDI*, pages 157–164, 1991.

[4] Shekhar Borkar. Thousand core chips — A technology perspective. In *DAC*, pages 746–749. IEEE, 2007.

[5] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell : a status report. In *DAMP '07 : Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM.

[6] Leland Chang, Yang-Kyu Choi, J. Kedzierski, N. Lindert, Peiqi Xuan, J. Bokor, Chenming Hu, and Tsu-Jae King. Moore's law lives on [cmos transistors]. *Circuits and Devices Magazine, IEEE*, 19(1) :35–42, Jan 2003.

[7] Murray I. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[8] Sylvain Conchon and Fabrice Le Fessant. Jocaml : Mobile agents for objective-caml. In *ASA/MA*, pages 22–29. IEEE Computer Society, 1999.

[9] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL*, pages 70–83, 1994.

[10] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL*, pages 113–123, 1993.

[11] Ulrich Drepper. What every programmer should know about memory. Technical report, November 2007.

[12] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP : Flexible, and efficient storage management for dynamically-typed languages. Technical Report TR 400, Indiana University, Computer Science Department, March 1994.

[13] Robert A. MacLachlan (ed). Design of CMU Common Lisp. `http://common-lisp.net/project/cmucl/doc/CMUCL-design.pdf`, 2003.

[14] R. Halstead. Implementation of multilisp : Lisp on a multiprocessor. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[15] Mark Jones and Paul Hudak. Implicit and explicit parallel programming in haskell. Technical Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, August 93.

[16] Donald E. Knuth. Interview with Donald Knuth. Web page at `http://www.informit.com/articles/article.aspx?p=1193856`, 2008.

[17] E. Ulrich Kriegel. A conservative garbage collector for an eulisp to ASM/C compiler. OOPSLA 1993 Workshop on Memory Management and Garbage Collection, September 1993.

[18] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. In Bernhard Gramlich, editor, *FroCoS*, volume 3717 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2005.

[19] Sanjiv Shah and Mark Bull. OpenMP - openMP. In *SC*, page 13. ACM Press, 2006.

[20] Guy Lewis Steele. Data representations in PDP-10 MACLISP. Report A. I. MEMO 420, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, 1977.

[21] Herb Sutter. The free lunch is over : a fundamental turn toward toward concurrency. *Dr. Dobb's Journal*, March 2005.

Annexe

$$[\text{Prog}_1]\ \frac{\Gamma, \sigma \rhd \underline{f_0} \ \rightarrow_f \ \Gamma', \sigma' \rhd \widetilde{f'_0}}{\Gamma, \sigma, (f_1 \ \ldots \ f_n) \rhd \underline{f_0} \ \rightarrow_p \ \Gamma', \sigma', (f_1 \ \ldots \ f_n) \rhd \widetilde{f'_0}}$$

$$[\text{Prog}_2]\ \frac{}{\Gamma, \sigma, (f_1 \ \ldots \ f_n) \rhd \overline{f'_0} \ \rightarrow_p \ \Gamma, \sigma, (f_2 \ \ldots \ f_n) \rhd \underline{f_1}}$$

$$[\text{Form}_1]\ \frac{\Gamma, \emptyset \ \vdash \ \sigma \rhd \underline{e_1} \ \rightarrow_e \ \sigma' \rhd \widetilde{e'_1}}{\Gamma, \sigma \rhd \underline{e_1} \ \rightarrow_f \ \Gamma, \sigma' \rhd \widetilde{e'_1}}\ e_1 \text{ n'a pas le car } \underline{\texttt{define}}$$

$$[\text{Form}_2]\ \frac{\Gamma, \emptyset \ \vdash \ \sigma \rhd \underline{e_1} \ \rightarrow_e \ \sigma' \rhd \widetilde{e'_1}}{\Gamma, \sigma \rhd \underline{\texttt{(define}\ \underline{x}\ \underline{e_1}\texttt{)}} \ \rightarrow_f \ \Gamma, \sigma' \rhd \underline{\texttt{(define}\ \underline{x}\ \widetilde{e'_1}\texttt{)}}}$$

$$[\text{Form}_3]\ \frac{}{\Gamma, \sigma \rhd \underline{\texttt{(define}\ \underline{x}\ \overline{e'_1}\texttt{)}} \ \rightarrow_f \ \Gamma[x \mapsto \overline{e'_1}], \sigma \rhd \overline{\texttt{\#t}}}$$

$$[\text{Atome}]\ \frac{}{\Gamma, \rho \ \vdash \ \sigma \rhd \underline{a} \ \rightarrow_e \ \sigma \rhd \overline{a}}\ a \text{ est un atome et n'est pas un symbole}$$

$$[\text{Variable}_\rho]\ \frac{}{\Gamma, \rho[x \mapsto \overline{e}] \ \vdash \ \sigma \rhd \underline{x} \ \rightarrow_e \ \sigma \rhd \overline{e}}\ x \text{ est un symbole}$$

$$[\text{Variable}_\Gamma]\ \frac{}{\Gamma[x \mapsto \overline{e}], \rho \ \vdash \ \sigma \rhd \underline{x} \ \rightarrow_e \ \sigma \rhd \overline{e}}\ x \text{ n'est pas liée dans } \rho$$

$$[\text{Quote}]\ \frac{}{\Gamma, \rho \ \vdash \ \sigma \rhd \underline{\texttt{(quote}\ \underline{e}\texttt{)}} \ \rightarrow_e \ \sigma \rhd \overline{e}}$$

$$[\text{Conditionnelle}_1]\ \frac{\Gamma, \rho \ \vdash \ \sigma \rhd \underline{e_1} \ \rightarrow_e \ \sigma' \rhd \widetilde{e'_1}}{\Gamma, \rho \ \vdash \ \sigma \rhd \underline{\texttt{(if}\ \underline{e_1}\ \underline{e_2}\ \underline{e_3}\texttt{)}} \ \rightarrow_e \ \sigma' \rhd \underline{\texttt{(if}\ \widetilde{e'_1}\ \underline{e_2}\ \underline{e_3}\texttt{)}}}$$

$$[\text{Conditionnelle}_2]\ \frac{}{\Gamma, \rho \ \vdash \ \sigma \ \rhd \underline{\texttt{(if}\ \overline{\texttt{\#t}}\ \underline{e_2}\ \underline{e_3}\texttt{)}} \ \rightarrow_e \ \sigma \rhd \underline{e_2}}$$

$$[\text{Conditionnelle}_3]\ \frac{}{\Gamma, \rho \ \vdash \ \sigma \ \rhd \underline{\texttt{(if}\ \overline{\texttt{\#f}}\ \underline{e_2}\ \underline{e_3}\texttt{)}} \ \rightarrow_e \ \sigma \rhd \underline{e_3}}$$

$$[\text{Abstraction}]\ \frac{}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\texttt{lambda}\ \underline{(\underline{x_1}\ \ldots\ \underline{x_n})}\ \underline{e})} \ \rightarrow_e\ \sigma \rhd \overline{\mathcal{C}(\rho, (x_1\ \ldots\ x_n), \underline{e})}}$$

$$[\text{Application}_1]\ \frac{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{e_i} \ \rightarrow_e\ \sigma' \rhd \widetilde{e'_i} \qquad \widetilde{e_0} \notin \{\underline{\texttt{if}}, \underline{\texttt{lambda}}, \underline{\texttt{let}}, \underline{\texttt{quote}}, \underline{\texttt{future}}\}}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\widetilde{e_0}\ \ldots\ \underline{e_i}\ \ldots\ \widetilde{e_n})} \ \rightarrow_e\ \sigma' \rhd \underline{(\widetilde{e_0}\ \ldots\ \widetilde{e'_i}\ \ldots\ \widetilde{e_n})}}\ 0 \leq i \leq n$$

$$[\text{Application}_2]\ \frac{\Gamma, \rho_1[x_1 \mapsto \overline{e'_1}, \ldots, x_n \mapsto \overline{e'_n}] \ \vdash\ \sigma \rhd \underline{e} \ \rightarrow_e\ \sigma' \rhd \widetilde{e'}}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\overline{\mathcal{C}(\rho_1, (x_1\ \ldots\ x_n), \underline{e})}\ \overline{e'_1}\ \ldots\ \overline{e'_n})} \ \rightarrow_e\ \sigma' \rhd \underline{(\overline{\mathcal{C}(\rho_1, (x_1\ \ldots\ x_n), \widetilde{e'})}\ \overline{e'_1}\ \ldots\ \overline{e'_n})}}$$

$$[\text{Application}_3]\ \frac{}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\overline{\mathcal{C}(\rho_1, (x_1\ \ldots\ x_n), \overline{e'})}\ \overline{e'_1}\ \ldots\ \overline{e'_n})} \ \rightarrow_e\ \sigma \rhd \overline{e'}}$$

$$[\text{Primitive}]\ \frac{p = cons \ \text{ ou } \ \overline{e_i} \neq \overline{\mathcal{F}(_)} \text{ pour tout } i}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\overline{\mathcal{P}(p)}\ \overline{e_1}\ \ldots\ \overline{e_n})} \ \rightarrow_e\ \sigma \rhd \overline{e}}\ p(e_1, \ldots, e_n) = e$$

$$[\text{Let}]\ \frac{}{\Gamma, \rho \ \vdash\ \sigma \ \rhd \underline{(\texttt{let}\ \underline{(\underline{(\underline{x_1}\ \underline{e_1})}\ \ldots\ \underline{(\underline{x_n}\ \underline{e_n})})}\ \underline{e})} \ \rightarrow_e\ \sigma \ \rhd \underline{(\underline{(\texttt{lambda}\ \underline{(\underline{x_1}\ \ldots\ \underline{x_n})}\ \underline{e})}\ \underline{e_1}\ \ldots\ \underline{e_n})}}$$

$$[\text{Fork}]\ \frac{}{\Gamma, \rho \ \vdash\ \sigma \rhd \underline{(\texttt{future}\ \underline{e})} \ \rightarrow_e\ \sigma[t \mapsto (\rho, \underline{e})] \rhd \overline{\mathcal{F}(t)}}\ t \text{ fraîche}$$

$$[\text{Inachevé}]\ \frac{\Gamma, \rho_1 \ \vdash\ \sigma \rhd \underline{e_1} \ \rightarrow_e\ \sigma' \rhd \widetilde{e'_1}}{\Gamma, \rho \ \vdash\ \sigma[t \mapsto (\rho_1, \underline{e_1})] \rhd \widetilde{e} \ \rightarrow_e\ \sigma'[t \mapsto (\rho_1, \widetilde{e'_1})] \rhd \widetilde{e}}$$

$$[\text{Achevé}]\ \frac{}{\Gamma, \rho \ \vdash\ \sigma[t \mapsto (\rho_1, \overline{e'_1})] \rhd \mathrm{C}[\overline{\mathcal{F}(t)}] \ \rightarrow_e\ \sigma[t \mapsto (\rho_1, \overline{e'_1})] \rhd \mathrm{C}[\overline{e'_1}]}\ \text{pour tout contexte C[.]}$$

Vérification d'invariants pour des systèmes spécifiés en logique de réécriture

Vlad Rusu[1] & Manuel Clavel[2,3]

1: Inria Rennes Bretagne-Atlantique, Rennes, France
rusu@irisa.fr
2: Universidad Complutense, Madrid, Spain
clavel@sip.ucm.es
3: IMDEA Software Institute, Madrid, Spain
clavel@imdea.es

Résumé

Nous présentons une approche basée sur la preuve inductive de théorèmes pour vérifier des invariants de systèmes spécifiés en *logique de réécriture*, un langage de spécification formelle implémenté dans l'outil Maude. Un invariant est une propriété qui est vraie dans tous les états atteignables à partir d'une certaine classe d'états initiaux. Notre approche consiste à coder les propriétés d'invariance de la logique de réécriture en *logique équationnelle avec appartenance*, une sous-logique de la logique de réécriture également implémentée dans Maude. Ce codage nous permet ensuite de prouver les propriétés d'invariance à l'aide d'un assistant de preuve disponible pour la logique équationnelle de Maude. Nous montrons que notre codage est correct, pour un sous-ensemble bien identifié (et suffisant en pratique) de systèmes et de propriétés d'invariance, et illustrons notre approche sur une version à n processus de l'algorithme Bakery.

1. Introduction

La logique de réécriture (*Rewriting Logic* [1], ou RL dans la suite de cet article) est un langage de spécification formelle dans lequel la dynamique d'un système est exprimée à l'aide de *règles de réécriture* qui agissent sur l'*état* du système, lui-même défini dans une *logique équationnelle* sous-jacente à la logique de réécriture. De nombreux auteurs ont mis en évidence l'adéquation de la logique de réécriture pour modéliser de nombreux types de systèmes dynamiques, comme par exemple les réseaux actifs [2], les sytèmes biologiques [3], ou la sémantique des langages de programmation [4]. Plusieurs systèmes implémentent des versions de cette logique : Maude [5], Elan [6], et CAFEOBJ [7].

La logique équationnelle sous-jacente à la logique de réécriture de Maude est la logique équationnelle avec appartenance (*Membership Equational Logic* [8], ou MEL dans la suite de cet article), qui permet d'exprimer, en plus des équations usuelles entre termes, *l'appartenance* d'un terme à une sorte. Notre approche pour la preuve inductive d'invariants utilise de telles appartenances, qui nous permettent de définir la sorte des termes accessibles à partir d'une classe de termes initiaux dans RL.

Le système Maude [5] donne une syntaxe concrète aux logiques MEL et RL, et fournit des outils qui permettent d'exécuter ces spécifications, de les analyser, et de les vérifier. Parmi ces ces outils, on trouve un explorateur exhaustif (à profondeur bornée) de l'ensemble des termes accessibles à partir d'un terme initial, et un *model checker* pour des propriétés exprimées dans la logique temporelle linéaire [9]. Ces deux outils sont limités, par nature, à des systèmes finis. Certaines classes de systèmes infinis peuvent aussi être vérifiées, en passant par des *abstractions équationnelles* [10] qui réduisent un

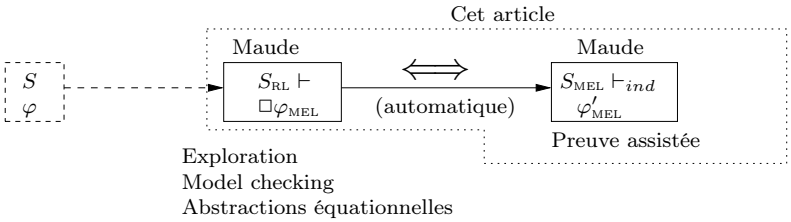


FIGURE 1 – Notre approche dans le contexte des outils de vérification du système Maude.

système infini vers un système fini, donc vérifiable par *model checking*. Cependant, ces abstractions ne préservent pas les propriétés d'invariance, en général, mais seulement dans des cas particuliers [10].

Les limites de la vérification automatisée justifient notre approche, basée sur la preuve assistée de théorèmes, pour prouver des propriétés d'invariance de systèmes dynamiques spécifiés en Maude. Notre approche consiste à traduire, de manière automatique, les propriétés d'invariance de spécifications en RL vers des propriétés inductives de spécifications MEL. Nous démontrons que la traduction est correcte, en ce sens qu'elle transforme un invariant qui est *vrai* sur une spécification RL donnée, vers une propriété (inductivement) *vraie* sur la théorie MEL obtenue par traduction. Ceci permet la vérification des invariants en utilisant des prouveurs inductifs pour MEL tel que l'outil ITP [11].

L'approche proposée complète donc la panoplie d'outils de vérification disponible pour Maude. Elle peut être utilisée en combinaison avec ces autres outils (cf. Figure 1). Une utilisation typique est la suivante : étant donné un système dynamique S et une propriété φ sur l'état du système, on commence par les formaliser, respectivement, par une spécification RL S_{RL} et par une formule de MEL φ_{MEL}. Ensuite, pour vérifier que φ_{MEL} est un invariant de S_{RL} (partant d'un ensemble d'états initiaux donnés) l'utilisateur peut, d'une part, essayer de falsifier l'invariance $S_{\text{RL}} \vdash \Box\varphi_{\text{MEL}}$ en utilisant le *model checker* de Maude, ou plus simplement l'exploration bornée des termes accessibles ; ou bien, l'utilisateur peut essayer de prouver la propriété avec le model checker, éventuellement en passant par des abstractions équationnelles, ou encore en utilisant notre approche basée sur la preuve assistée.

Le reste de l'article est organisé de la façon suivante. Dans la section 2 nous présentons brièvement les logiques MEL et RL. Nous définissons ensuite la notion d'*invariant φ (exprimé en logique* MEL*) d'une spécification $\mathcal{R}$ (exprimée en* RL*) à partir d'une classe initiale d'états t_0(représentée par un terme t_0, qui peut contenir des variables libres)*, ainsi : $\varphi(t)$ est prouvable dans le modèle standard de la sous-théorie MEL de $\mathcal{R}$, pour tous terme t accessible depuis t_0 par *réécriture close à la racine* dans $\mathcal{R}$. Nous notons l'invariance par $\langle \mathcal{R}, t_0 \rangle \vdash \Box\varphi$. La réécriture close à la racine veut dire, intuitivement, que le filtrage se fait à la racine, et que les substitutions utilisées sont closes, pour toutes les variables apparaîssant dans le terme réécrit et dans la régle de réécriture. Nous donnons des arguments en faveur de l'adéquation de ce type de réécriture pour exprimer la dynamique de systèmes exprimés en RL, et plus particulièrement lorsque les états initiaux sont exprimés par des termes à variables libres et les règles de réécriture comportent des variables supplémentaires en partie droite et/ou dans les conditions associées. Ces aspects sont essentiels lorsqu'on souhaite modéliser des systèmes à nombre paramétrique de processus, tel que l'algorithme Bakery à n processus qui constitue notre application.

Le cœur de notre approche est présenté en section 3. D'abord, on définit une traduction automatique qui prend une théorie RL $\mathcal{R}$ et un terme t_0, et qui engendre une théorie MEL $\mathcal{M}(\mathcal{R}, t_0)$ qui enrichit la sous-théorie MEL de $\mathcal{R}$ avec une nouvelle sorte, appelée *Reachable*, définie inductivement à partir des règles de $\mathcal{R}$ et du terme t_0. Ensuite, on démontre que les affirmations *être de la sorte Reachable dans $\mathcal{M}(\mathcal{R}, t_0)$* et *être accessible dans $\mathcal{R}$ à partir de t_0 par réécriture close à la racine* sont équivalentes. Puis, nous donnons une définition alternative à l'invariance dans $\mathcal{R}$ à partir de t_0, comme suit : *$\varphi(t)$ est vraie dans le modèle standard (appelé aussi le modèle initial) de la théorie $\mathcal{M}(\mathcal{R}, t_0)$ construite précédemment, et ce, pour tous les termes clos ayant la sorte Reachable dans le modèle initial*. Enfin, nous prouvons que les deux définitions d'invariance sont équivalentes ; l'avantage de la deuxième définition est qu'elle permet l'emploi de prouveurs inductifs pour la logique MEL, tel que l'outil ITP. L'utilisation de notre approche pour prouver l'exclusion mutuelle de l'algorithme Bakery à n processus est décrite en section 4. Nous concluons et présentons des travaux connexes et futurs en section 5.

2. Logique équationnelle avec appartenance et logique de réécriture

Nous présentons ici brièvement ces deux logiques ; des présentations plus complètes existent [1, 8]. Une *signature* en MEL est un tuple (K, Σ, S), où K est un ensemble de *kinds*[1], Σ est une famille indexée sur $K^* \times K$ de symboles de fonction : $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$, et $S = \{S_k\}_{k\in K}$ est une famille, indexée par K, de *sortes* - où S_k représente l'ensemble des sortes du kind k. Une signature (K, Σ, S) est souvent notée par sa seule composante Σ ; T_Σ est alors l'ensemble de termes clos sur Σ. Etant donné un ensemble de paires $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ de variables, chacune munie de son kind, $T_\Sigma(X)$ représente alors l'ensemble des termes dont les variables libres sont dans l'ensemble X. De manière similaire, $T_{\Sigma,k}$ et $T_{\Sigma,k}(X)$ représentent, respectivement, l'ensemble des termes clos du kind k, et l'ensemble des termes clos du kind k aux variables libres dans X. Une formule atomique de MEL dans la signature (K, Σ, S) est soit une *équation* $t = t'$, avec $t, t' \in T_{\Sigma,k}(X)$ et $k \in K$, soit *un axiome d'appartenance* $t : s$, avec $t \in T_{\Sigma,k}(X)$, s une sorte dans S_k, et $k \in K$. Une *phrase* dans MEL est une clause de Horn, implicitement universellement quantifiée sur les variables apparaîssant dans les formules atomiques :

$$(\forall X)t = t' \ \textit{if}\ C, \text{ or} \tag{1}$$
$$(\forall X)t : s \ \textit{if}\ C \tag{2}$$

où la *condition* C est de la forme suivante, avec I, J deux ensembles finis d'indices :

$$\bigwedge_{i\in I}(u_i = v_i) \wedge \bigwedge_{j\in J}(w_j : s_j)$$

Les phrases de la forme (1) sont appelées *équations conditionnelles*, et les phrases de la forme (2) sont des *appartenances conditionnelles*. Une équation ou une appartenance sont inconditionnelles lorsqu'elles sont atomiques. Une théorie en MEL est une couple $\mathcal{M} = (\Sigma, E)$ composé d'une signature MEL Σ et d'un ensemble E de phrases sur Σ. La logique MEL possède un système de déduction, donné en Définition 1 ci-dessous, qui est complet et cohérent [8], en ce sens qu'une formule atomique φ est déductible dans une théorie (Σ, E) - ce qu'on note par $(\Sigma, E) \vdash \varphi$, ou plus simplement $E \vdash \varphi$, si et seulement φ est sémantiquement valide : elle est vraie dans tous les modèles de la théorie (Σ, E)

Définition 1 *Soit une théorie $\mathcal{M} = (\Sigma, E)$ et une formule atomique e, de la forme $(\forall X)t = t'$ ou $(\forall X)t : s$. Alors, e est déductible dans $\mathcal{M}$, noté $\mathcal{M} \vdash e$ ou simplement $E \vdash e$ lorsque la signature résulte du contexte, si e peut être obtenue en appliquant les règles suivantes :*

1. *Reflexivité :* $\frac{t\in T_\Sigma(X)}{E\vdash(\forall X)t=t}$
2. *Appartenance :* $\frac{E\vdash(\forall X)t'=t \quad E\vdash(\forall X)t:s}{E\vdash(\forall X)t':s}$
3. *Symmétrie :* $\frac{E\vdash(\forall X)t'=t}{E\vdash(\forall X)t=t'}$
4. *Transitivité :* $\frac{E\vdash(\forall X)t_1=t_2 \quad E\vdash(\forall X)t_2=t_3}{E\vdash(\forall X)t_1=t_3}$
5. *Congruence :* $\frac{f\in\Sigma_{k_1,\ldots,k_n,k} \quad t_1,\ldots t_n\in T_{\Sigma_{k_i}}(X) \quad pour\ i\in[1,n] \quad t'_i\in T_{\Sigma_{k_i}}(X) \quad E\vdash(\forall X)t_i=t'_i}{E\vdash(\forall X)f(t_1,\ldots,t_i,\ldots,t_n)=f(t_1,\ldots,t'_i,\ldots,t_n)}$
6. *Remplacement*$_1$ $\frac{(\forall X)t=t'\ if\ C \ \in E \quad \sigma:X\mapsto T_\Sigma(Y) \quad E\vdash(\forall Y)C\sigma}{E\vdash(\forall Y)t\sigma=t'\sigma}$
7. *Remplacement*$_2$ $\frac{(\mu)\ (\forall X)t:s\ if\ C \ \in E \quad \sigma:X\mapsto T_\Sigma(Y) \quad E\vdash(\forall Y)C\sigma}{E\vdash(\forall Y)t\sigma:s}$

1. Nous gardons le nom anglais faute de traduction satisfaisante. La raison d'être des kinds (par rapport aux sortes) est de contenir des termes syntaxiquement corrects mais sémantiquement incorrects, comme par exemple l'"entier" 1/0.

où $\sigma : X \mapsto T_\Sigma(Y)$ *sont des substitutions préservant les kinds, et pour une condition* C : $\bigwedge_{i\in I}(\forall X)(u_i = v_i) \wedge \bigwedge_{j\in J}(\forall X)(w_j : s_j)$, *la notation* $E \vdash (\forall Y)C\sigma$ *est une abbréviation de la conjonction* $\bigwedge_{i\in I} E \vdash (\forall Y)(u_i\sigma = v_i\sigma) \wedge \bigwedge_{j\in J} E \vdash (\forall Y)(w_j\sigma : s_j)$.

Il convient de remarquer que la déductibilité (et, de manière équivalente, la validité sémantique) ne sont pas toujours la notion de vérité attendue. En effet, beaucoup de propriétés interessantes ne sont pas vraies dans *tous* les modèles d'une théorie MEL, mais seulement dans certains, dont le *modèle initial* [8]. Dans ce modèle, les sortes sont interprétées comme les plus petits ensembles satisfaisant les axiomes (équations et appartenances), et l'égalité est la plus petite congruence satisfaisant ces mêmes axiomes. On écrit $E \vdash_{ind} (\forall X)\varphi$ pour dire que la phrase φ est vraie dans le modèle initial de (Σ, E).

Exemple 1 *Soit la théorie* MEL *NAT constituée de :*

- $K_{NAT} = \{Nat?\}$.
- $\Sigma_{NAT_{(\lambda,Nat?)}} = \{0\}$.
- $\Sigma_{NAT_{(Nat?,Nat?)}} = \{s\}$.
- $\Sigma_{NAT_{w,Nat?}} = \emptyset$, *pour* $w \notin \{\lambda, Nat?\}$.
- $S_{Nat?} = \{Nat\}$.
- $E_{NAT} = \{0 : Nat, (\forall N)\, s(N) : Nat \ \ if \ \ N : Nat\}$.

Le modèle initial de NAT est l'ensemble des nombres naturels $\mathbb{N}$. *Mais ils existe de nombreux autres modèles, parmi lesquels, par exemple, il y a tous les ensembles d'entiers modulo* n, *pour tout* $n \geq 2$.

Une théorie en logique de réécriture (ou simplement, une théorie de réécriture, ou de RL) est un tuple $\mathcal{R} = (K, \Sigma, S, E, R)$, où (K, Σ, S, E) est une théorie MEL, et R est en ensemble de règles de réécriture

$$(\rho) \quad (\forall X)\ l \to r \quad if \quad C \tag{3}$$

où la condition C est de la forme $\bigwedge_{i\in I}(u_i = v_i) \wedge \bigwedge_{j\in J}(w_j : s_j)$ avec I et J des ensemble finis d'indices. Dans sa forme la plus générale [12], la logique RL permet également des *réécritures dans les conditions des règles* et des *arguments gelés*, deux constructions avancées de RL qu'on ne considère pas ici.

Définition 2 *Etant donnée une théorie* $\mathcal{R} = (\Sigma, E, R)$ *et deux termes* $t, t' \in T_\Sigma(X)$, *on dit que* t' *est accessible* $\mathcal{R}$ *depuis* t *par réécriture à la racine, noté par* $\mathcal{R} \vdash (\forall X)t \longrightarrow t'$, *si la formule* $(\forall X)t \longrightarrow t'$ *peut être obtenue en appliquant les rgles suivantes :*

1. *Reflexivité :* $\frac{t\in T_\Sigma(X)}{\mathcal{R}\vdash(\forall X)t\longrightarrow t}$
2. *Transitivité :* $\frac{\mathcal{R}\vdash(\forall X)t_1\longrightarrow t_2 \quad \mathcal{R}\vdash(\forall X)t_2\longrightarrow t_3}{\mathcal{R}\vdash(\forall X)t_1\longrightarrow t_3}$
3. *Egalité :* $\frac{E\vdash(\forall X)t=u \quad \mathcal{R}\vdash(\forall X)u\longrightarrow u' \quad E\vdash(\forall X)u'=t'}{\mathcal{R}\vdash(\forall X)t\longrightarrow t'}$
4. *Remplacement :* $\frac{(\rho)\ (\forall X)\ l\longrightarrow r \ if\ C\in R \quad \sigma:X\mapsto T_\Sigma(Y) \quad E\vdash(\forall Y)C\sigma}{\mathcal{R}\vdash(\forall Y)l\sigma\longrightarrow r\sigma}$

On notera que dans sa forme la plus générale [12], la logique comporte aussi une règle de Congruence. Eliminer cette règle revient à forcer la réécriture à la racine.

Nous introduisons maintenant la réécriture *close* à la racine. Intuitivement, lorsqu'on réécrit de cette manière un terme $(\forall X)t$ par une règle $(\forall Y)l \to r \ if \ C$, on imposera que le terme t soit d'abord transformé en un terme clos $\sigma(t)$ au moyen d'une substitution close σ de ses variables ; et seulement ensuite, le terme $\sigma(t)$ pourra être réécrit à la racine, en imposant que le filtrage avec le membre gauche l de la règle se fasse par une substitution close de *toutes* les variables apparaissant dans l, r, et C.

Définition 3 *Etant donnée une théorie* $\mathcal{R} = (\Sigma, E, R)$ *et deux termes* $t \in T_\Sigma(X)$ *et* $t' \in T_\Sigma$, *on dira que* t' *est accessible depuis* t *par réécriture close à la racine, noté* $\mathcal{R} \vdash\!\downarrow (\forall X)t \longrightarrow t'$, *s'il existe une substitution close* $\sigma : X \mapsto T_\Sigma$ *et une dérivation* $\mathcal{R} \vdash \sigma(t) \longrightarrow t'$ *à la racine (cf. Définition 2) dans laquelle toutes les applications de la règle de* Remplacement *utilisent des substitutions closes.*

Exemple 2 *Pour illustrer notre notion de réécriture close à la racine, considerons le terme* $s(W)$ *dans la théorie NAT et la règle* $s(X) \to s(s(Y))$*. Le terme* $s(W)$ *peut être réécrit par substitution close à la racine, en une étape, en* $s(s(0))$ *: on transforme* $s(W)$ *en* $s(0)$ *par la substitution close* $W \leftarrow 0$ *; et on réécrit* $s(0)$ *en* $s(s(0))$ *avec la règle, au moyen de la substitution close* $X \leftarrow 0, Y \leftarrow 0$*.*

Il convient de remarquer que la réécriture habituelle (non-close), du terme donné avec la règle donnée, contiendra forcément des séquences de termes non-clos, e.g., $s(W), s(s(W)), \ldots, s^n(W), \ldots$*, alors que la réécriture close engendre seulement des termes clos. C'est la raison principale pour laquelle la réécriture close est préférable à la réécriture standard pour exprimer la dynamique de systèmes modélisés en* RL *(et, spécialement, lorsque l'ensemble d'états initiaux est représenté par un terme non-clos et/ou les règles de réécriture comportent des variables supplémentaires en partie droite et/ou dans la condition) : la réécriture close engendre seulement des termes clos, qui dénotent des états, alors que les termes non-clos ne dénotent pas des états, mais des ensembles d'états. Les systèmes paramétriques, tel l'algorithme Bakery à n processus que nous présentons en section 4, utilisent naturellement des terme non-clos pour dénoter l'ensemble de leurs états initiaux, ainsi que des règles de réécriture avec variables suppplémentaires en partie droite et dans les conditions pour exprimer leur dynamique.*

L'exemple suivant est simple (terme initial clos, pas de variable supplementaire a droite des règles) ; il nous sert comme exemple de motivation, et illustre aussi l'adéquation de la réécriture *à la racine* pour exprimer la dynamique de (nombreux) systèmes exprimés en RL.

Exemple 3 *Le problème des lecteurs-écrivains est un problème classique en programmation concurrente. Il s'agit d'assurer l'accès en lecture/écriture à un fichier, de telle manière qu'il y ait exclusion mutuelle entre lecteurs et écrivains et entre écrivains. L'accès simultané de plusieurs lecteurs au fichier est permis. Une spécification de ce système en* RL*, appelée READERS-WRITERS, inclut la déclaration d'une sorte State et du kind correspondant* $[State]$ *pour décrire les états du système ainsi que la déclaration d'un constructeur* $\langle .,. \rangle$ *qui prend deux nombres naturels (le nombre de lecteurs et le nombre d'écrivains, respectivement) et qui rend un State. L'évolution du système est décrite par les règles de réécriture suivantes :*

$$R_{READERS\text{-}WRITERS} = \begin{cases} \langle 0,0 \rangle \to \langle 0, s(0) \rangle, \\ \langle R, s(W) \rangle \to \langle R, W \rangle, \\ \langle R, W \rangle \to \langle s(R), W \rangle \ \textit{if} \ W = 0, \\ \langle s(R), W \rangle \to \langle R, W \rangle \end{cases}$$

La première règle spécifie que depuis un état $\langle 0,0 \rangle$ *dans lequel in n'y a ni lecteur, ni écrivain, le système peut accepter un écrivain. La seconde règle permet à un écrivain de quitter le fichier ; la troisième permet d'accepter un lecteur supplémentaire, pourvu qu'il n'y ait pas d'écrivains ; et la quatrième règle permet à un lecteur de sortir du fichier. Cette spécification à un nombre d'états qui est infini (le nombre de lecteurs pouvant croître indéfiniment par la troisième règle). Nous allons illustrer notre approche (présentée dans la section suivante) pour vérifier l'exclusion mutuelle entre lecteurs et écrivains.*

Enfin, remarquons le fait que la réécriture dans READERS-WRITERS s'effectue toujours à la racine : les parties gauches des règles filtrent des termes du kind $[State]$*, et les réécrivent en termes du même kind ; et ces termes n'ont pas de sous-termes stricts du kind* $[State]$ *; par conséquent, la réécriture ne peut avoir lieu en dessous de la racine. Ce type de réécriture est souvent suffisant en pratique pour spécifier des systèmes en* RL*, comme remarqué également par d'autres auteurs [13].*

L'invariance pour les systèmes spécifiés en RL

Nous continuons cette section par une définition de la notion d'invariance pour des systèmes spécifiés en RL. Intuitivement, un prédicat sur les états d'un système est un invariant si le prédicat est vrai dans tous les états du système qui sont accessibles à partir d'une certaine classe d'états initiaux. Afin de formaliser cette notion pour les systèmes spécifiés en RL nous devons préciser les aspects suivants :

1. quels sont les états et la dynamique du système ?
2. comment spécifier les prédicats sur les états ?
3. quand ces prédicats sont-ils *vrais* dans un état donné ?

Nous proposons que les états du système soient représentés par des termes clos d'un certain kind $[State]$, et que la dynamique du système soit définie par la réécriture close à la racine, partant d'un ensemble d'états initiaux représentés par un terme t_0, qui peut avoir des variables libres, et qui est également du kind $[State]$. Nous avons discuté ci-dessus l'adéquation de la réécriture close à la racine (Définition 3) pour exprimer la dynamique de systèmes exprimés en RL, tout particulièrement lorsque le terme initial comporte des variables libres et/ou les règles de réécriture comportent des variables supplémentaires en partie droite et/ou dans la condition, dont nous verrons l'utilité en section 4.

Concernant les prédicats d'état, ils seront formalisés par des phrases de la logique MEL de la forme $(\forall x : [State])(\forall Y)\varphi$, i.e., des phrases ayant une variable x du kind $[State]$, ainsi qu'optionnellement d'autres variables Y, telles que $x \notin Y$. La variable x du kind $[State]$ fait qu'on puisse parle de φ comme prédicate d'*état* ; les autres variables servent à quantifier universellement sur les composantes de l'état ; nous en verrons l'utilité également en section 4.

Enfin, un prédicat d'état $(\forall x : [State])(\forall Y)\varphi$ est vrai dans un état t lorsque le le prédicat $(\forall Y)\varphi(t/x)$, obtenu à partir de φ en substituant la variable x par le terme t, est vrai dans le modèle initial de la sous-théorie MEL E de la spécification RL $\mathcal{R}$ du système : $E \vdash_{ind} (\forall Y)\varphi(t/x)$.

En conclusion, lorsqu'un système est spécifié en RL, on formalise l'idée qu'un invariant est un prédicat d'état qui est vrai dans touts les états du système qui sont accessibles à partir d'une certaine classe d'états initiaux, ainsi :

Définition 4 *Soient $\mathcal{R} = (K, S, \Sigma, E, R)$ une théorie de réécriture, $[State] \in K$ un kind, $(\forall X)t_0 \in T_{\Sigma,k}(X)$ un terme, et $(\forall x : [State])(\forall Y)\varphi$ une phrase* MEL. *Alors, φ est un invariant de $\mathcal{R}$ en partant de t_0, noté $\langle \mathcal{R}, t_0 \rangle \vdash \Box\varphi$, si pour tout $t \in T_{\Sigma,[State]}$, $\mathcal{R} \vdash\!\downarrow (\forall X)t_0 \longrightarrow t$ implique $E \vdash_{ind} (\forall Y)\varphi(t/x)$.*

Exemple 4 *Considérons la spécification READERS-WRITERS donnée dans l'exemple 3. Supposons qu'un prédicat $>$ ait été défini dans la sous-théorie* MEL *NAT, et supponsons aussi que les deux accesseurs standard fst et snd aux éléments d'une paire sont définis. Nous décrivons l'exclusion mutuelle entre lecteurs et écrivains par le prédicat mutex, défini par les équations suivantes :*

$$\begin{array}{l}(\forall\, x : [State])mutex(x) = true \ \ if \ \ fst(x) = 0\\ (\forall\, x : [State])mutex(x) = true \ \ if \ \ snd(x) = 0\\ (\forall\, x : [State])mutex(x) = false \ \ if \ \ fst(x) > 0 \wedge \ snd(x) > 0\end{array}$$

On remarque que mutex est une formule atomique de MEL, *qui contient une seule variable x du kind $[State]$. L'invariance de mutex sur la théorie READERS-WRITERS partant du terme initial (clos) $\langle 0, 0\rangle$ est notée $\langle READERS\text{-}WRITERS, \langle 0, 0\rangle\rangle \vdash \Box mutex$ et définie par le fait que, pour tout terme clos t du kind $[State]$, on a $[READERS\text{-}WRITERS \vdash\!\downarrow \langle 0, 0\rangle \longrightarrow t]$ implique $[NAT \vdash_{ind} mutex(t/x)]$.*

Falsification automatique d'invariants

Avant d'aborder la preuve interactive d'invariants dans la section suivante, nous concluons cette section sur la falsification automatique d'invariants. La preuve et la falsification sont évidemment des techniques complémentaires, et disposer d'une technique automatique pour falsifier un invariant est utile avant de démarrer une preuve interactive qui, dans le cas d'une propriété fausse, n'aboutira pas.

Nous allons considerer des énoncés d'invariance $\langle \mathcal{R}, t_0 \rangle \vdash \Box\varphi$ dans lesquelles la théorie $\mathcal{R}$ est *exécutable*[2] et tels que le terme intial t_0 est clos et le prédicat φ a une seule variable, qui est

2. Cette condition exige, entre autres, qu'il n'y ait pas de variable supplémentaire en partie droite et dans les conditions ; le système peut quand même avoir un nombre infini d'états, cf. [5] pour une définition complète.

du kind [*State*], comme dans l'exemple $\langle \mathit{READERS\text{-}WRITERS}, \langle 0,0\rangle\rangle \vdash \Box \mathit{mutex}$. Nous montrons que si ces contraintes sont satisfaites, il existe une procédure qui est correcte et complète pour la falsification d'invariants. Cette procédure est implémentée par la commande `search` de l'outil Maude. Les contraintes sur le terme initial et le prédicat d'état peuvent être éliminées, au prix de la perte de complétude de la procédure, qui reste cependant correcte. Nous présentons ici les aspects de la commande `search` qui sont utiles pour nos besoins ; une description complète est disponible dans [5].

Soit un prédicat d'état $(\forall x : [State])\varphi$ où $\varphi \triangleq \varphi_0$ *if* $\varphi_1 \ldots \varphi_n$, et φ_i sont des équations ou des appartenances atomiques, pour $i = 0, \ldots, n (n \in \mathbb{N})$. Nous utilisons des commandes `search` de la forme

$$\texttt{search}\ t_0\ \texttt{=>*}\ x\ \texttt{such that}\ \varphi_1(x) \wedge \ldots \wedge \varphi_n(x) \wedge \neg\varphi_0(x) \tag{4}$$

où t_0 est un terme clos, et est x une variable, tous les deux du kind [*State*]. La commande ci-dessus effectue une recherche en largeur des termes accessibles par réécriture depuis t_0 [3]. La commande *termine avec succès* si elle retourne au moins un terme accessible depuis t_0 et satisfaisant les conditions de la clause `such that`. Le fait que la commande `search` (4) soit une procédure correcte et complète pour la falsification d'invariants est précisé comme suit :

Observation 1 *Soit une propriété d'invariance de la forme* $\langle \mathcal{R}, t_0\rangle \vdash \Box\varphi$ *telle que : la théorie* RL $\mathcal{R}$ *est exécutable ; le terme* t_0 *est clos ; et le prédicat d'état* φ *a une seule variable libre, qui est du kind* [*State*]. *Alors, la command* `search` *(4) se termine avec succès si et seulement si* $\langle \mathcal{R}, t_0\rangle \not\vdash \Box\varphi$.

Cette observation est vraie pour les raisons suivantes : les contraintes d'exécutabilité garantissent que, d'une part, les conditions de la clause `such that` sont décidables et que, d'autre part, la commande `search` trouve tout état *accessible* depuis t_0 en un temps fini [5] ; et par ailleurs, lorsque le terme initial est clos et la spécification est exécutable (donc elle ne contient pas de variables supplémentaires en parties droites et dans les conditions de règles) la réécriture à la racine est close par construction.

Exemple 5 *La spécification* RL *READERS-WRITERS et le prédicat mutex de l'exemple 4 satisfont les contraintes de l'observation 1 ci-dessus. On peut tenter de falsifier l'énoncé* $\langle \mathit{READERS\text{-}WRITERS}, \langle 0,0\rangle\rangle \vdash \Box \mathit{mutex}$ *ainsi :* `search` $\langle 0,0\rangle$ `=>*` x `such that` $\neg \mathit{mutex}(x)$.

La commande `search` *ci dessus ne termine pas, car l'énoncé* $\langle \mathit{READERS\text{-}WRITERS}, \langle 0,0\rangle\rangle \vdash \Box \mathit{mutex}$ *est vrai, comme nous le démontrons dans la section suivante. Un exemple de falsification réussie est celle du prédicat* same-number, *défini par* same-number$(x) = \mathit{true}$ *if* $\mathit{fst}(x) = \mathit{snd}(x)$, same-number$(x) = \mathit{false}$ *if* $\mathit{fst}(x) < \mathit{snd}(x)$, *et* same-number$(x) = \mathit{false}$ *if* $\mathit{fst}(x) > \mathit{snd}(x)$. *Dans ce cas, la commande* `search` *retourne* $x = \langle 0,0\rangle$, *qui constitue un contre-exemple à l'invariance.*

Enfin, remarquons que, bien que la commande `search` exige un terme initial t_0 clos et une seule variable dans le prédicat d'état φ dont on cherche à falsifier l'invariance, on peut, en sacrifiant la complétude, instancier "à la main" les variables supplémentaires éventuelles dans t_0 and φ avant de lancer la commande `search`, sans compromettre la correction de la procédure de falsification.

3. Preuve assistée pour les propriétés d'invariance

Dans cette section nous présentons notre approche basée sur la preuve assistée pour prouver des propriétés d'invariance de spécifications en RL. D'abord, nous définissons une traduction automatique qui prend en entrée une théorie RL $\mathcal{R}$ et un terme t_0 et engendre une théorie MEL $\mathcal{M}(\mathcal{R}, t_0)$, qui enrichit la sous-théorie MEL de $\mathcal{R}$ avec une nouvelle sorte, appelée *Reachable*, et avec des axiomes d'appartenance à cette sorte. Ensuite, nous démontrons qu'*être accessible dans* $\mathcal{R}$ *à partir de* t_0 est

3. Nous supposons que la réécriture se fait à la racine ; sinon, il est toujours possible d'*encapsuluer* le terme et les règles pour forcer la réécriture à la racine [13].

équivalent à *être de la sorte Reachable dans la théorie* MEL $\mathcal{M}(\mathcal{R}, t_0)$. Puis, nous montrons un corrolaire de ce résultat, qui dit que pour tout prédicat d'état $(\forall x : [State])(\forall X)\varphi$, l'invariance $\langle \mathcal{R}, t_0 \rangle \vdash \Box\varphi$ est équivalente au fait que l'implication $(\forall x : [State])(\forall X)(x : Reachable \Rightarrow \varphi)$ est vraie dans le modèle initial de la théorie $\mathcal{M}(\mathcal{R}, t_0)$. Enfin, en remarquant que la preuve par induction est *correcte* (en anglais : *sound*) dans les modèle initiaux des théories MEL, nous établissons la correction de notre approche pour prouver des invariants par preuve inductive, grâce à l'équivalence établie ci-dessus.

Définition 5 *Soit une théorie* RL $\mathcal{R} = (K, \Sigma, S, E, R)$ *munie d'une sorte State* $\in S$ *au kind [State]* $\in K$*, et un terme* $t_0 \in T_{\Sigma,[State]}(X)$*. On note par* $\mathcal{M}(\mathcal{R}, t_0)$ *la théorie* MEL $(K_{\mathcal{M}(\mathcal{R},t_0)}, \Sigma_{\mathcal{M}(\mathcal{R},t_0)}, S_{\mathcal{M}(\mathcal{R},t_0)}, E_{\mathcal{M}(\mathcal{R},t_0)})$ *construite ainsi :*

- $K_{\mathcal{M}(\mathcal{R},t_0)} = K$
- $\Sigma_{\mathcal{M}(\mathcal{R},t_0)} = \Sigma$
- $S_{\mathcal{M}(\mathcal{R},t_0)} = S \cup S'_{[State]}$ *avec* $S'_{[State]} = S_{[State]} \cup \{Reachable\}$ *et* $Reachable \notin S$
- $E_{\mathcal{M}(\mathcal{R},t_0)} = E \cup \{(\forall X)t_0 : Reachable\} \cup \{\mu(\rho) | (\rho) \in R\}$*, où* $\mu((\rho)(\forall X)\ l \to r\ if\ C))$ *dénote l'axiome d'appartenance* $(\forall X)\ r : Reachable\ if\ \ l : Reachable \wedge C$*.*

Dans la définition précédente, on remarque l'axiome d'appartenance $\{(\forall X)t_0 : Reachable\}$ ainsi que les axiomes d'appartence $\mu(\rho)$, un par régle de réécriture ρ dans $\mathcal{R}$. L'axiome pour t_0 dit que t_0 est accessible, et l'axiome $\mu(\rho)$ "inverse" la règle ρ, afin d'exprimer le fait que, dans la réécriture, la partie droite de ρ est accessible dès lors que sa partie gauche est accessible et que sa condition est satisfaite.

Exemple 6 *Considérons le problème des lecteurs-écrivains donné dans l'exemple 3. La théorie* MEL $\mathcal{M}(READERS\text{-}WRITERS, \langle 0,0 \rangle)$ *contient les éléments suivants.*

- $K_{\mathcal{M}(READERS\text{-}WRITERS\langle 0,0\rangle)} = K_{READERS\text{-}WRITERS}$.
- $\Sigma_{\mathcal{M}(READERS\text{-}WRITERS,\langle 0,0\rangle)} = \Sigma_{READERS\text{-}WRITERS}$.
- $S_{\mathcal{M}(READERS\text{-}WRITERS,\langle 0,0\rangle)} = S_{READERS\text{-}WRITERS} \cup S'_{[State]}$,
  avec $S'_{[State]} = S_{READERS\text{-}WRITERS_{[State]}} \cup \{Reachable\}$
- $E_{\mathcal{M}(READERS\text{-}WRITERS,\langle 0,0\rangle)} = E_{READERS\text{-}WRITERS} \cup E'$*, où*

$$E' = \begin{cases} \langle 0,0 \rangle : Reachable\ , \\ \langle 0, s(0) \rangle : Reachable\ if\ \ \langle 0,0 \rangle : Reachable\ , \\ \langle R, W \rangle : Reachable\ if\ \ \langle R, s(W) \rangle : Reachable\ , \\ \langle s(R), W \rangle : Reachable\ if\ \langle R, W \rangle : Reachable \wedge\ W = 0, \\ \langle R, W \rangle : Reachable\ if\ \langle s(R), W \rangle : Reachable\ . \end{cases}$$

Théorème 1 *Soit une théorie* RL $\mathcal{R} = (K, \Sigma, S, E, R)$ *munie d'une sorte State* $\in S$ *au kind [State]* $\in K$*, et un terme* $t \in T_{\Sigma,[State]}(X)$*. Alors, pour tout terme clos* $t' \in T_{\Sigma,[State]}$*, on a l'équivalence* $\mathcal{R} \vdash\!\downarrow (\forall X)t \longrightarrow t'$ *si et seulement si* $\mathcal{M}(\mathcal{R}, t) \vdash t' : Reachable$*.*

La preuve est donnée en Annexe. L'idée est de mettre en relation les pas de réécriture close à la racine avec une règle donnée ρ dans la théorie RL $\mathcal{R}$, et les pas de déduction avec l'axiome d'appartenance $\mu(\rho)$ dans la théorie MEL $\mathcal{M}(\mathcal{R}, t)$. On utilise également le fait que la sorte *Reachable* est "neuve" donc elle ne peut pas influencer la déduction dans la sous-théorie MEL (Σ, E) de $\mathcal{R}$.

Observation 2 *Il est intéressant de remarquer l'importance de la réécriture* close *dans le théorème 1. En effet, on a le contre-exemple suivant pour la réécriture non close : soit la théorie de réécriture constituée du kind Foo, de la constante a du kind Foo, de la fonction* $f : Foo \mapsto Foo$*, et d'aucune sorte, équation, appartenance, et règle de réécriture. Soit* $t_0 = f(x)$ *le terme initial. Alors, d'après les règles de la définition 2, seul* $f(x)$ *est atteignable. Et pourtant, la théorie obtenue par la transformation* $\mathcal{M}()$ *permet de déduire* $f(a) : Reachable$*, en appliquant* Remplacement_2 *(définition 1) avec l'appartenance* $f(x) : Reachable$ *et la substitution* $\sigma : x \leftarrow a$*. La réécriture non-close ne permettrait donc pas le*

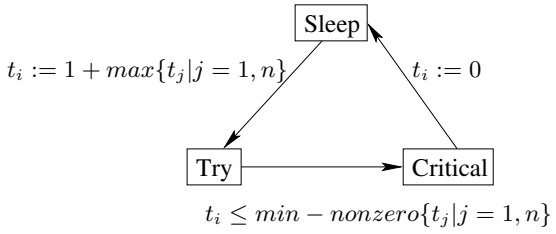


FIGURE 2 – Le i-ème processus dans l'algorithme Bakery à n processus

théorème 1. En revanche, pour la réécriture close, le "contre-exemple" ci-dessus n'en est plus un : on a bien que $f(a)$ est atteignable en utilisant la substitution $\sigma : \ x \leftarrow a$ dans la définition 3.

Pour le corrolaire suivant, rappelons que $\vdash_{ind}$ dénote la vérité dans un modèle initial.

Corollaire 1 *Soit une théorie* RL $\mathcal{R} = (K, \Sigma, S, E, R)$ *munie d'une sorte State $\in S$ au kind [State] $\in K$, et soit un predicat d'état $(\forall x : [State], \forall X)\varphi$. Alors, on a l'équivalence $\langle \mathcal{R}, t_0 \rangle \vdash \Box\varphi$ si et seulement si $\mathcal{M}(R, t_0) \vdash_{ind} (\forall x : [State])(\forall X)(x : Reachable \Rightarrow \varphi)$.*

La preuve est donnée en Annexe. Elle utilise le théorème 1, ainsi que quelques résultats standard sur les modèles initiaux et, à nouveau, le fait que la sorte *Reachable* est "neuve" dans $\mathcal{M}(\mathcal{R}, t)$ et par conséquent n'influence pas la vérité dans le modèle initial de (Σ, E).

Exemple 7 *Afin de prouver $\langle READERS\text{-}WRITERS, \langle 0, 0 \rangle \rangle \vdash \Box mutex$ nous allons prouver $\mathcal{M}(READERS\text{-}WRITERS, \langle 0, 0 \rangle) \vdash_{ind} (\forall x)(x : Reachable \Longrightarrow mutex(x) = true)$. Nous utilisons pour cela l'assistant de preuve* ITP. *L'induction sur la sorte Reachable engendre cinq sous-buts. Le premier d'entre eux représente la base de l'induction, et correspond à l'appartenance au terme intial à la sorte Reachable. Les quatres autres sous-buts représentent les pas d'induction, qui correspondent aux quatre règles de réécriture définissant la dynamique du système. Ici, tous les sous-buts sont prouvés automatiquement avec la commande* `auto` *de l'outil* ITP, *qui invoque la réécriture automatique et les procédures de décision pour l'arithmétique linéaire. Bien entendu, les preuves ne sont pas toujours aussi automatiques. Dans la section suivante nous donnons un exemple dans lequel l'invariant principal a besoin d'*invariants auxiliaires *afin de passer les étapes d'induction de la preuve avec* ITP.

4. Vérification de l'algorithme Bakery à n processus

L'algorithme Bakery à n processus est un système paramétrique, qui consiste en un nombre $n \geq 2$ de processus identiques. Chacun des processus peut être dans l'une des localités *Sleep*, *Try*, ou *Critical*, et chacun possède un ticket que lui seul peut écrire, et que tous les processus peuvent lire (cf. Fig. 2). Dans l'état initial, tous les processus sont dans *Sleep* et tous les tickets t_i valent 0.

En allant de *Sleep* à *Try*, le i-ème processus affecte son ticket à la valeur maximum des tickets plus un. La condition pour entrer en section critique - dans notre modélisation, dans la localité *Critical* - pour le i-éme processus, est que son ticket soit inférieur un égal au *minimum non-zéro* du multi-ensemble de tickets des processus. Le minimum non-zéro d'un multi-ensemble non vide de nombre naturels est défini par by *min-nonzero(S)* $= 0$ si S contient seulement (un certain nombre de fois) l'élément 0 ; et *min-nonzero*$(S) = min(S \setminus \{0\})$ sinon, où *min* est le minimum habituel d'un multi-ensemble non vide de nombres naturels, et $S \setminus \{0\}$ est le multi-ensemble obtenu en enlevant à S toutes les instances de 0. Enfin, lorsqu'un processus retourne dans la localité *Sleep* il remet son ticket à zéro.

La propriété d'exclusion mutuelle veut dire qu'un seul processus peut être à la fois en section critique. Pour la prouver, d'abord informellement, puis formellement avec notre approche, nous avons besoin de trois lemmes :

– les tickets des processus qui se trouvent en *Try* ou *Critical* sont *strictement positifs* ;

– les tickets des processus qui se trouvent en *Try* ou *Critical* sont *distincts deux à deux* ;
– les tickets des processus qui se trouvent en *Critical* valent le minimum non-zéro des tickets.

Le premier lemme est vrai car, lorsque les processus entrent dans *Try*, ils affectent à leur ticket une valeur strictement positive - le maximum des tickets plus un - et le fait d'entrer dans *Critical* ne change pas la valeur des tickets. Le second lemme est vrai car, en entrant dans *Try*, chaque processus affecte à son ticket une valeur *nouvelle* - le maximum des tickets plus un, et le fait d'entrer dans *Critical* ne change pas la valeur des tickets. Le troisième lemme est vrai car (a) la condition pour entrer dans *Critical* est d'avoir un ticket plus petit ou égal au minimum non-zéro des tickets, (b) le minimum non-zéro d'un multi-ensemble non vide de tickets non-nuls (tel que garanti par le premier lemme) est égal au minimum habituel du multi-ensemble, et (c) il n'y a pas de ticket plus petit que le minimum. Enfin, la propriété d'exclusion mutuelle est vraie car, en plus, le minimum d'un multi-ensemble non vide de valeurs distinctes deux à deux (tel que garanti par le deuxième lemme), est unique ; par conséquent, un seul processus, détenteur de cet unique ticket, peut être en section critique à un moment donné.

Spécification de l'algorithme en RL. Nous commençons par la définition de la sorte des *états locaux*, qui sont des paires $\langle L, N\rangle$ formées d'une localité L et d'un nombre naturel N (le ticket), ainsi que des accesseurs $().loc$ et $().tic$ aux états locaux, comme suit

$$\begin{array}{l}(\forall\, L, N)(\langle L, N\rangle : \mathit{LocalState}\ \mathit{if}\ \ L : \mathit{Loc}\ \ \wedge\ \ N : \mathit{Nat})\\ (\forall\, L, N)\langle L, N\rangle.\mathit{loc} = L\\ (\forall\, L, N)\langle L, N\rangle.\mathit{tic} = N\end{array}$$

Ensuite, nous définissons la sorte des états, qui sont soit des états locaux, soit une concaténation d'un état local et d'un état, séparés par ";" :

$$\begin{array}{l}(\forall\, LS)(LS : \mathit{State}\ \mathit{if}\ LS : \mathit{LocalState})\\ (\forall\, LS, ST)((LS; ST) : \mathit{State}\ \mathit{if}\ \ LS : \mathit{LocalState} \wedge ST : \mathit{State})\end{array}$$

Puis, la taille *dim()* d'un état est définie par le nombre d'états locaux qui y participent, et les accesseurs et modificateurs pour les états sont définis ainsi : $ST[i]$ rend le i-ème état local de l'état ST, et $ST\ \mathit{with}[i] := LS$, rend une copie de l'état ST dans lequel le i-ème état local est remplacé par LS. Nous définissons aussi le maximum $\mathit{max}(ST)$ et le minimum non-zéro $\mathit{min\text{-}nonzero}(ST)$ des tickets.

Enfin, les transitions de l'algorithme sont exprimées par trois règles de réécriture, qui sont paramétrées par l'indice i du processus qui change d'état. La règle (5) exprime la transition du processus i de *Sleep* à *Try*, la règle (6), celle de *Try* à *Critical*, et la règle (6), celle de *Critical* à *Sleep*.

$$\begin{array}{l}(\forall\, i, ST)ST \rightarrow (ST\ \mathit{with}[i] := \langle \mathit{Try}, 1 + \mathit{max}(ST)\rangle)\\ \qquad \mathit{if}\ \ i : \mathit{Int}\ \ \wedge i \geq 0 \wedge\ \ i < \mathit{dim}(ST)\ \ \wedge\ \ ST[i].\mathit{loc} = \mathit{Sleep}\end{array} \tag{5}$$

$$\begin{array}{l}(\forall\, i, ST)ST \rightarrow (ST\ \mathit{with}[i] := \langle \mathit{Critical}, ST[i].\mathit{tic}\rangle)\\ \qquad \mathit{if}\ \ i : \mathit{Int}\ \ \wedge i \geq 0 \wedge\ \ i < \mathit{dim}(ST)\ \ \wedge\ \ ST[i].\mathit{loc} = \mathit{Try}\end{array} \tag{6}$$

$$\begin{array}{l}(\forall\, i, ST)ST \rightarrow (ST\ \mathit{with}[i] := \langle \mathit{Sleep}, 0\rangle)\\ \qquad \mathit{if}\ \ i : \mathit{Int}\ \ \wedge i \geq 0 \wedge\ \ i < \mathit{dim}(ST)\ \ \wedge\ \ ST[i].\mathit{loc} = \mathit{Critical}\end{array} \tag{7}$$

On remarque la présence de la variable supplémentaire i en partie droite de la règle et dans la condition. Sans ces variables il ne serait pas aisé de spécifier les systèmes paramétriques comme celui que nous étudions ici.

En choisissant un nombre fixe de processus (e.g., 2) et en dupliquant les règles afin de remplacer la variable i par les valeurs constantes que cette variable peut prendre, on peut exécuter le protocole, chercher des contre-exemples pour la propriété d'exclusion mutuelle (8) donnée ci dessous, etc.

Transformation de l'algorithme en MEL et vérification. Afin de procéder à la vérification au moyen de notre approche, nous transformons d'abord automatiquement l'algorithme en une théorie MEL comme indiqué dans la section précédente. Les appartenances conditionnelles associés aux règles (5),(6), et (7) sont les suivantes :

$$(\forall\, i, ST)(ST\ with[i] := \langle Try, 1 + max(ST)\rangle) : Reachable\ \ if\ \ ST : Reachable \\ \wedge\ i : Int \wedge i \geq 0 \wedge i < dim(ST)\ \ \wedge\ \ ST[i].loc = Sleep$$

$$(\forall\, i, ST)(ST\ with[i] := \langle Critical, ST[i].tic\rangle) : Reachable\ \ if\ \ ST : Reachable \\ \wedge\ i : Int \wedge i \geq 0 \wedge i < dim(ST)\ \ \wedge\ \ ST[i].loc = Try$$

$$(\forall\, i, ST)(ST\ with[i] := \langle Sleep, 0\rangle) : Reachable\ \ if\ \ ST : Reachable \\ \wedge\ i : Int \wedge i \geq 0 \wedge i < dim(ST)\ \ \wedge\ \ ST[i].loc = Critical$$

Ensuite, l'ensemble des états initiaux dans lequel se trouve l'algorithme est un nombre infini d'états locaux, ayant la localité *Sleep* et la valeur 0 pour le ticket, et tous ces états sont accessibles :

$$Init(0) = \langle Sleep, 0\rangle \\ (\forall n) Init(s(n)) = \langle Sleep, 0\rangle; Init(n) \\ (\forall n) Init(n) : Reachable\ \ if\ \ n : Nat$$

Puis, nous formalisons la propriété d'exclusion mutuelle comme le théorème suivant :

$$(\forall\, ST, i, j)\, ST : Reachable \wedge i : Int \wedge i \geq 0 \wedge i < dim(ST) \wedge j : Int \\ \wedge\ j \geq 0 \wedge j < dim(ST) \wedge\ \ ST[i] = Critical \wedge ST[j] = Critical \\ \Longrightarrow i = j \tag{8}$$

Le théorème dit que dans tous les états accessibles, si deux états locaux en positions i et j sont tous les deux dans *Critical* alors $i = j$. On remarque la présence et l'utilité des variables universellement quantifiées i et j, qui apparaîssent dans la propriété en plus de la variable ST dénotant l'état.

Enfin, nous prouvons formellement la propriété d'exclusion mutuelle avec l'assistant de preuve ITP, au moyen de trois lemmes, de la même manière que la preuve informelle donnée en début de section. Le premier lemme dit que dans toutes les localités sauf *Sleep*, les tickets sont strictement positifs :

$$(\forall\, ST, i)\, ST : Reachable\ \ \wedge\ \ i : Int\ \ \wedge i \geq 0 \wedge\ \ i < dim(ST) \wedge\ \ \neg isSleep(ST[i].loc) \\ \Longrightarrow ST[i].tic > 0$$

Le deuxième lemme dit que dans toutes les localités sauf *Sleep* les tickets sont distincts :

$$(\forall\, ST, i, j)\, ST : Reachable \wedge i : Int \wedge i \geq 0 \wedge i < dim(ST) \wedge j : Int \wedge j \geq 0 \\ \wedge j < dim(ST) \wedge\ \ i < j \wedge \neg isSleep(ST[i].loc) \wedge \neg isSleep(ST[j].loc) \\ \Longrightarrow ST[i].tic \neq ST[j].tic$$

Le troisième lemme que dans la localité *Critical*, les tickets sont inférieur ou égaux au minimum non-zéro des tickets :

$$(\forall\, ST, i)\, ST : Reachable\ \ \wedge\ \ i : Int \wedge i \geq 0 \wedge i < dim(ST) \wedge ST[i].loc = Critical \\ \Longrightarrow ST[i].tic \leq min\text{-}nonzero(ST)$$

Les preuves avec ITP du théorème et des trois lemmes suivent toutes le même canevas. Elles commencent par une induction sur la sorte *Reachable*, qui engendre quatre sous-buts : un pour les états initiaux, et trois pour les trois transitions du système. Le pas de base (pour les états initiaux) est résolu par la commande `auto`, qui invoque la réécriture et les procédures de décision du prouveur. Les pas inductifs sont des combinaisons de `auto`, d'analyses par cas, et de preuves d'invariants auxiliaires.

Nous avons également eu à prouver plusieurs lemmes sur *tous* les états (pas seulement sur les états accessibles, donc, des lemmes qui techniquement ne sont pas des invariants). Ces lemmes formalisent certaines relations entre les fonctions que nous avons définies, par exemple, le fait que le minimum non-zéro d'un multi-ensemble non vide de valeurs non nulles est égal au minimum du multi-ensemble, et inférieur ou égal au maximum du multi-ensemble. Les surces ITP pour cet exemple sont disponibles à l'URL `http ://www.irisa.fr/vertecs/Equipe/Rusu/itp/mutex-n`.

5. Conclusion, travaux connexes, et perspectives

L'exploration de l'ensemble des états accessibles et le *model checking* pour les systèmes finis, les abstractions pour réduire les systèmes infinis vers des systèmes finis, et la preuve interactive pour les systèmes infinis sont toutes des techniques de vérification bien connues. Pour la logique de réécriture, toutes ces approches sauf la dernière existent dans l'environnement Maude. Notre contribution a l'ambition de combler ce manque. L'approche que nous proposons est basée sur une traduction automatique d'un sous-ensemble significatif de la logique de réécriture RL (sans réécritures dans les conditions des règles, sans arguments gelés, et dont la réécriture se fait de manière close à la racine), et de propriétés d'invariance exprimées en logique équationnelle avec appartenance MEL, vers des propriétés inductives exprimées dans cette dernière logique. Grâce à cette traduction, dont nous prouvons la correction, nous pouvons utiliser des assistants de preuve pour la logique équationnelle, tel que l'outil ITP, pour prouver des invariants sur des systèmes exprimés en logique de réécriture.

L'approche est illustrée sur l'algorithme Bakery à n processus. Les résultats sont encourageants, et ce, malgré le fait que l'assistant de preuve ITP est encore à l'état de prototype. Une trait distinctif de notre approche est qu'elle s'intègre naturellement avec les autres outils disponibles pour Maude, comme l'exploration énumérative, le *model checking*, et les abstractions équationnelles (figure 1).

L'approche proposée présente bien entendu des limites. Comme toutes les approches basées sur la preuve assistée, elle exige un utilisateur dédié et expert. D'après notre expérience, l'expertise s'obtient de manière incrémentale durant la vérification d'un exemple donné, et l'utilisateur bénéficie du retour de l'outil ITP lorsqu'il échoue à prouver un invariant donné pour cause d'information insuffiante : cette information se trouve dans le sous-but laissé non prouvé et dans le contexte dans lequel la preuve échoue. En examinant ces retours de l'outil, il est relativement aisé de poser un lemme qui, une fois prouvé, permettrait de clore la preuve du sous-but qui a engendré le problème. La difficulté principale de type d'approche réside dans le nombre de lemmes auxiliaires ; le pendant de l'explosion combinatoire en *model checking* est ici l'explosion du nombre de lemmes auxiliaires.

Travaux connexes. Nous présentons quelques travaux connexes dans la multitude des travaux existants. L'équipe CafeOBJ du Japan Advanced Institute of Science and Technology (JAIST) propose une approche pour prouver des invariants de systèmes dynamiques exprimés comme des *Observational Transition Systems* (OTS), les invariants étant des prédicats d'état sur les OTS. Ensuite, l'OTS et prédicat peuvent être représentés dans plusieurs formalismes (cf. Figure 3) : CafeOBJ et Coq, pour la preuve [14, 15] ; Maude, pour la falsification d'invariants [16] ; et SMV, pour le *model checking* [17].

Le plus proche, dans les travaux de l'équipe CafeOBJ, de ce que nous proposons est la preuve d'invariants en CafeOBJ et leur falsification en Maude. Pour ce qui est de la preuve, leur approche consiste à coder l'invariance comme des prédicats sur des paires d'états, exprimant le fait que les différentes transitions du système *préservent* le prédicat d'état dont on souhaite prouver l'invariance. Ensuite, la réduction équationnelle de CafeOBJ tenter d'établir automatiquement cette préservation.

Des opérations typiquement effectuées par un assistant de preuve, telles que le remplacement des variables universellement quantifiées par des constantes "neuves" et la décomposition en cas, sont laissées ici à l'utilisateur. Ceci constitue une source d'erreurs dans des preuves de taille importante. Pour ce qui est de la falsification d'invariants, comme cette possibilité n'existe pas en CafeOBJ, les

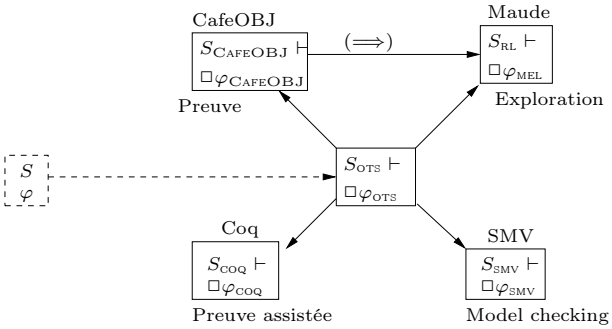


FIGURE 3 – *Observational Transition Systems* : représentation en CafeOBJ et en d'autres outils.

auteurs ont choisi de traduire les OTS en Maude et d'utiliser l'exploration énumérative. Globalement, la différence principale entre ce que nous proposons et la proposition de l'équipe CAFEOBJ réside dans le fait que nous restons dans un seul environnement et formalisme (celui de Maude, et de la logique de réecriture/équationnelle avec appartenance), ce qui nous permet de nous appuyer sur une sémantique commune dans les diverses activités de vérification (Figure 1). Ceci contraste fortement avec l'approche de l'équipe CAFEOBJ, qui utilise plusieurs outils, aux sémantiques *a priori* différentes (Figure 3). Leur approche soulève un problème de cohérence globale : pour un problème de vérification donné, obtient-on des réponses cohérentes avec l'ensemble des outils connectés ? Un début de réponse est donné dans [18] pour la correction de la traduction entre CAFEOBJ et Maude.

Dans l'article [12] les auteurs présentent un codage exhaustif de RL dans MEL. Leur codage traite toute la logique RL, y compris les aspects que nous avons choisi de laisser de coté comme les réécritures dans les conditions des règles, les arguments gelés, et la réécriture générale (pas forcément à la racine et pas forcément close). Leur codage est assez complexe [4], et leur but est d'étudier la sémantique et la théorie de la preuve de RL en terme des notions correspondantes de MEL. Notre objectif est différent : obtenir un codage d'un sous-ensemble de la logique RL, qui soit suffisant en pratique pour exprimer la dynamique de nombreux systèmes *et de leurs invariants exprimés en* MEL, et qui soit assez simple pour être utilisable dans une démarche de preuve interactive assistée. Un codage *simple* est essentiel pour que l'utilisateur retrouve, à travers le codage, les propriétés qu'il tente de prouver.

La vérification de systèmes paramétriques est en général indécidable. Par conséquent, leur vérification automatique est limitée à des sous-classes, ou est approchée. Parmi les nombreuses approches automatiques, il y a le *regular model checking* [19], où l'état du système est modélisé par un langage régulier, et la relation de transition, par un transducteur ; les *abstractions de comptage* [20], dans lesquelles seul le nombre de processus dont le contrôle se trouve dans une localité donnée est mémorisé, la vérification étant approchée par cette abstraction ; et la *génération d'invariants* observés sur des instances particulière du système, qu'on généralise et prouve sur le système paramétrique.

Les perspectives ouvertes par cette recherche sont d'utiliser effectivement la combinaison d'approches formelles existantes dans Maude, y compris la nôtre, pour vérifier une application réaliste, et montrer que l'utilisateur gagne à utiliser les différentes méthodes ensemble plutôt qu'une seule.

Références

[1] N. Martí-Oliet and J. Meseguer. Rewriting logic : roadmap and bibliography. *TCS*, 285(2) :121–154, 2002.

[2] J. Meseguer, P. C. Ölveczky, M. O. Stehr, and C. L. Talcott. Maude as a wide-spectrum framework for formal modeling and analysis of active networks. In *DANCE*, pages 494–510. IEEE Comp. Soc., 2002.

4. Pour l'accessibilité, leur codage comprend, *pour chaque kind* k de la théorie RL à coder, un nouveau kind k', quatre nouvelles sortes, et quatre nouvelles opérations définies par sept nouvelles équations dans la théorie MEL résultante.

[3] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and M. K. Sönmez. Pathway logic : Symbolic analysis of biological signaling. In *Pacific Symposium on Biocomputing*, pages 400–412, 2002.

[4] J. Meseguer and G. Rosu. The rewriting logic semantics project. *TCS*, 373(3) :213–237, 2007.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[6] P. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, and C. Ringeissen. An overview of ELAN. *Electr. Notes Theor. Comput. Sci.*, 15, 1998.

[7] R. Diaconescu and K. Futatsugi. Logical foundations of CafeOBJ. *TCS*, 285(2) :289–318, 2002.

[8] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

[9] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. *Electr. Notes Theor. Comput. Sci.*, 71, 2002.

[10] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2003.

[11] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool : a tutorial. *J. Universal Computer Science*, 12(11) :1618–1650, 2006.

[12] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *TCS*, 360(1-3) :386–414, 2006.

[13] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to the verification of cryptographic protocols. In *WRLA*, volume 117 of *Electronic Notes in Theoretical Computer Science*, 2004.

[14] K. Futatsugi. Verifying specifications with proof scores in CafeOBJ. In *ASE*, pages 3–10. IEEE Comp. Soc., 2006.

[15] Kazuhiro Ogata and Kokichi Futatsugi. State machines as inductive types. *IEICE Transactions*, 90-A(12) :2985–2988, 2007.

[16] W. Kong, T. Seino, K. Futatsugi, and K. Ogata. A lightweight integration of theorem proving and model checking for system verification. In *APSEC*, pages 59–66. IEEE Comp. Soc., 2005.

[17] K. Ogata, M. Nakano, M. Nakamura, and K. Futatsugi. Chocolat/SMV : a translator from CafeOBJ to SMV. In *PDCAT*, pages 416–420. IEEE Comp. Soc., 2005.

[18] M. Nakamura, W. Kong, K. Ogata, and K. Futatsugi. A specification translation from behavioral specifications to rewrite specifications. *IEICE Transactions*, 91-D(5) :1492–1503, 2008.

[19] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *TCS*, 256(1-2) :93–112, 2001.

[20] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2002.

Annexe

Théorème 1 *Soit une théorie* RL $\mathcal{R} = (K, \Sigma, S, E, R)$ *munie d'une sorte State* $\in S$ *au kind [State]* $\in K$*, et un terme* $t \in T_{\Sigma,[State]}(X)$*. Alors, pour tout terme clos* $t' \in T_{\Sigma,[State]}$*, on a l'équivalence* $\mathcal{R} \vdash\downarrow (\forall X) t \longrightarrow t'$ *si et seulement si* $\mathcal{M}(\mathcal{R}, t) \vdash t'$: *Reachable.*

Preuve. ($\Rightarrow$) Par induction sur la longueur de la réécriture close $\mathcal{R} \vdash \sigma(t) \longrightarrow t'$ correspondant à $\mathcal{R} \vdash\!\downarrow (\forall X)t \longrightarrow t'$ (cf. définition 3). Ici, la longueur est le nombre d'applications de la règle de *Remplacement* dans la définition 2. Si la longueur est 0 alors $\sigma(t)$ et t' sont *égaux modulo* E (c'est à dire, $E \vdash \sigma(t) = t'$). En utilisant $(\forall X)t : \mathit{Reachable}$ dans $\mathcal{M}(\mathcal{R}, t)$ on obtient, avec les règles de déduction de la logique MEL (définition 1), $\mathcal{M}(\mathcal{R}, t) \vdash \sigma(t) : \mathit{Reachable}$ puis $\mathcal{M}(\mathcal{R}, t) \vdash t' : \mathit{Reachable}$.

Nous supposons l'implication vraie pour les dérivations de longueur n et la prouvons pour la longueur $n + 1$. *Modulo* l'égalité par les equations E, une dérivation de longueur $n + 1$ peut être décomposée en $\mathcal{R} \vdash \sigma(t) \longrightarrow t'' \longrightarrow t'$, pour un terme clos donné t'', tel que le dernier pas $\mathcal{R} \vdash t'' \longrightarrow t'$ soit une application du *Remplacement*, avec une règle (ρ) $(\forall X)l \to r$ *if* C et une substitution $\sigma' : X \mapsto T_\Sigma$.

1. alors, $t'' \equiv \sigma'(l)$, $t' \equiv \sigma'(r)$ et $E \vdash \sigma'(C) = \mathit{true}$, qui implique *a fortiori* $\mathcal{M}(\mathcal{R}, t) \vdash \sigma'(C) = \mathit{true}$. (On note par $\equiv$ l'égalité syntaxique entre deux termes).
2. par l'hypothèse d'induction, $\mathcal{M}(\mathcal{R}, t) \vdash t'' : \mathit{Reachable}$, i.e., $\mathcal{M}(\mathcal{R}, t) \vdash \sigma(l) : \mathit{Reachable}$
3. donc, on peut utiliser $\mathit{Replacement}_2$ (définition 1) avec $(\mu(\rho))$ $r : \mathit{Reachable}$ *if* $l : \mathit{Reachable} \wedge C$ (définition 5) et σ', et en conclure $\mathcal{M}(\mathcal{R}, t) \vdash \sigma'(r) : \mathit{Reachable}$, i.e., $\mathcal{M}(\mathcal{R}, t) \vdash t' : \mathit{Reachable}$.

($\Leftarrow$) Par induction sur la longueur de la dérivation $\mathcal{M}(\mathcal{R}, t) \vdash t' : \mathit{Reachable}$. Ici, la longueur est le nombre d'applications de $\mathit{Remplacement}_2$ (définition 1) avec la sorte $s = \mathit{Reachable}$. On remarque d'abord que $n \geq 1$, car, afin d'inférer l'appartenance à *Reachable*, on doit utiliser au moins une fois $\mathit{Remplacement}_2$ avec $s = \mathit{Reachable}$. Le pas de base $n = 1$ implique que, nécessairement, $\mathit{Remplacement}_2$ ait été utilisée avec l'appartenance $(\forall X)t : \mathit{Reachable}$ du terme initial. En effet, toutes les autres appartenances définissant *Reachable* sont récursives, e.g., elles nécessitent déjà une appartenance à *Reachable* pour être utilisées avec $\mathit{Remplacement}_2$. Par conséquent, il existe une substitution close σ telle que $t' \equiv \sigma(t)$, et ensuite on obtient trivialement $\mathcal{R} \vdash \sigma(t) \longrightarrow t'$ par *Réflexivité*.

On suppose l'implication *vraie* pour toutes les dérivations de longueur n et la prouvons pour la longueur $n+1$. *Modulo* les équations E, une preuve de longueur $n+1$ peut toujours être décomposée en une preuve $\mathcal{M}(R, t) \vdash t'' : \mathit{Reachable}$ de longueur n, pour un terme donné $t'' \in T_\Sigma$, suivie d'une application de $\mathit{Remplacement}_2$, avec une appartenance $(\mu(\rho))$ $r : \mathit{Reachable}$ *if* $l : \mathit{Reachable} \wedge C$ obtenue à partir de la règle (ρ) $(\forall X)l \to r$ *if* C. Alors, il existe une substitution close σ' telle que $t' \equiv \sigma'(r)$, $t'' \equiv \sigma'(l)$, et $\mathcal{M}(R, t) \vdash \sigma'(C) = \mathit{true}$. Puisque la sorte *Reachable* est "neuve" dans $\mathcal{M}(R, t)$, elle n'apparaît pas dans la condition C, et par conséquent on a aussi $E \vdash \sigma(C) = \mathit{true}$, et, par *Remplacement* on obtient $\mathcal{R} \vdash \sigma'(l) \longrightarrow \sigma'(r)$, i.e., $\mathcal{R} \vdash t'' \longrightarrow t'$. Par hypothèse d'induction, $\mathcal{R} \vdash\!\downarrow (\forall X)t \longrightarrow t''$ i.e., il existe une substitution close $\sigma : X \mapsto T_\Sigma$ telle qu'on ait une dérivation $\mathcal{R} \vdash \sigma(t) \longrightarrow t''$ où toutes les applications de *Remplacement* se font avec des substitutions closes. Comme la dérivation $\mathcal{R} \vdash t'' \longrightarrow t'$ que nous venons d'obtenir utilise aussi une substitution close - σ' - on obtient par *Transitivité* $\mathcal{R} \vdash \sigma(t) \longrightarrow t'$, et par la Définition 3, on en conclut $\mathcal{R} \vdash\!\downarrow (\forall X)t \longrightarrow t'$. □

Corollaire 1 *Soit une théorie* RL $\mathcal{R} = (K, \Sigma, S, E, R)$ *munie d'une sorte State* $\in S$ *au kind [State]* $\in K$*, et soit un predicat d'état* $(\forall x : [State], \forall X)\varphi$*. Alors, on a l'équivalence* $\langle \mathcal{R}, t_0 \rangle \vdash \Box\varphi$ *si et seulement si* $\mathcal{M}(R, t_0) \vdash_{ind} (\forall x : [State])(\forall X)(x : \mathit{Reachable} \Rightarrow \varphi)$.

Preuve. En partant de $\mathcal{M}(R, t_0) \vdash_{ind} (\forall x : [State])(x : \mathit{Reachable} \Rightarrow (\forall X)\varphi)$ et en utilisant le fait qu'une formule universellement quantifiée est vraie dans le modèle initial d'une théorie si et seulement si toutes les instances closes de la formule sont vraies dans ce même modèle, on obtient de manière équivalente $\forall t \in T_{\Sigma,[State]}.\, \mathcal{M}(R, t_0) \vdash_{ind} (t : \mathit{Reachable} \Rightarrow (\forall X)\varphi(t/x))$. Ensuite, en uilisant l'équivalence : $A \vdash (B \Rightarrow C)$ ssi ($A \vdash B$ implique $A \vdash C$), on obtient de manière équivalente

$$\forall t \in T_{\Sigma,[State]}.\, ([\mathcal{M}(R, t_0) \vdash t : \mathit{Reachable}] \ \mathit{implique}\ [\mathcal{M}(R, t_0) \vdash_{ind} (\forall X)\varphi(t/x)]) \tag{9}$$

Par le théorème 1, $\mathcal{M}(\mathcal{R}, t_0) \vdash t : \mathit{Reachable}$ est équivalent à $\mathcal{R} \vdash\!\downarrow (\forall X)t_0 \longrightarrow t$; et, puisque φ ne fait pas référence à la "nouvelle" sorte *Reachable*, $\mathcal{M}(\mathcal{R}, t_0) \vdash_{ind} (\forall X)\varphi(t/x)$ si et seulement $E \vdash_{ind} (\forall X)\varphi(t/x)$. Et par conséquent, l'implication (9) s'écrit de manière équivalente $\forall t \in T_{\Sigma,[State]}.$ $\mathcal{R} \vdash\!\downarrow (\forall X)t_0 \longrightarrow t$ *implique* $E \vdash_{ind} (\forall X)\varphi(t/x)$, qui est la définition 4 de l'invariance $\langle \mathcal{R}, t_0 \rangle \vdash \Box\varphi$. □

Un modèle de l'assistant à la preuve : PAF!

Séverine Maingaud*

*: *Laboratoire PPS,*
Université Paris Diderot - Paris 7
Case 7014
75205 PARIS Cedex 13
maingaud@pps.jussieu.fr

Résumé

Nous proposons un modèle du système PAF!, un assistant à la preuve de programme dédié au fragment fonctionnel de ML. En particulier, nous modélisons le prédicat de typage fort propre au système. De la correction de la logique du système vis-à-vis du modèle présenté ici nous déduisons la cohérence du formalisme, ainsi que la propriété de terminaison pour tous les programmes fortement typés (au sens du système).

1. Introduction

PAF! est un assistant à la preuve dédié au fragment fonctionnel de ML, développé par S. Baro et P. Manoury. Le but de cet assistant est de permettre à l'utilisateur de prouver l'adéquation d'un programme fonctionnel vis-à-vis d'une spécification donnée, sans restriction sur les formes de récursion utilisées, autorisant ainsi le raisonnement sur les fonctions partielles et/ou définies par récursion non structurelle. [1] Le moteur de preuve de PAF! est fondé sur une logique multi-sortée du second ordre à la AF2 dont les objets de base sont les programmes fonctionnels (avec récursion non gardée). Le principal ingrédient du langage de spécification est un prédicat de *typage fort,* $t \downarrow \tau$, qui exprime la correction forte du programme t vis-à-vis du type τ (laquelle implique notamment la terminaison de ce programme dans tous les contextes appropriés à son type). Dans [3], S. Baro et P. Manoury présentent une preuve de correction de ce système de typage renforcé restreint aux entiers naturels.

Nous présentons ici une preuve de cohérence d'une variante du formalisme (dans laquelle on a supprimé le mécanisme de définition de types inductifs et remplacé l'algèbre de types de ML par celle du système F) à l'aide d'un modèle de réalisabilité construit avec des techniques standard.

Les rôles du typage Le système PAF! tel qu'il est présenté dans [2] utilise deux mécanismes de typage des termes : le *typage statique* (inféré par la machine) et le *typage fort* (prouvé par l'utilisateur). Le typage statique de PAF! correspond au typage traditionnel de ML et sert à prévenir un certain nombre d'erreurs à l'exécution. La seule garantie offerte est qu'un programme bien typé dont l'exécution termine retourne une valeur du type attendu. Le typage statique de PAF! n'offre en revanche aucune garantie sur la terminaison des programmes, la bonne fondation des structures de données (listes cycliques) ou la présence d'exceptions non rattrapées (un problème que nous n'aurons pas à traiter dans la mesure où le calcul présenté ici est dépourvu d'un mécanisme d'exceptions).

1. Par ailleurs, l'implémentation actuelle de PAF! autorise les définitions inductives avec des occurrences négatives dans le type des constructeurs, avec tous les phénomènes de non terminaison qui y sont associés [6]. Nous ne traiterons pas ce cas dans le système présenté ici.

Dans PAF! le typage apparaît également sous la forme d'un prédicat de typage fort (dans les formules) dont la preuve est à la charge de l'utilisateur. Ce prédicat exprime non seulement l'absence d'erreur à l'exécution, mais aussi la terminaison sur une valeur du type voulu. Ici, la « force » doit s'entendre au sens de la garantie offerte et non au sens de la décidabilité de la relation. Le typage fort de PAF! ne doit donc pas être confondu avec le typage fort de ML, qui est un typage statique décidable n'offrant pas de garantie de terminaison. En outre, le prédicat de typage fort accepte des programmes ordinairement refusés par le typage statique, tels que `if true then 42 else false` (de type `nat` au sens fort). En revanche, il sera impossible de typer au sens fort des programmes tels que `let rec boucle x = boucle x in boucle` (dans notre langage le `rec` est une macro pour le point fixe), qui sont pourtant acceptés par la plupart des systèmes de typage statique.

Un des buts de cet article est de montrer que sur le plan théorique le typage statique n'est pas indispensable pour raisonner sur la terminaison des programmes ; le typage fort de PAF! est à lui seul suffisant. Pour cette raison nous allons présenter une version étendue et simplifiée du système PAF! dans laquelle on a débranché le typage statique. Dans le système présenté ici, les types apparaîtront uniquement comme des invariants logiques utilisés par le prédicat de typage fort. Même si dans l'implémentation il est préférable de conserver le typage statique qui reste une aide précieuse pour le programmeur, notre approche montre que la preuve de programme peut prendre en compte les constructions qui passent outre le typage statique, comme la primitive `Obj.magic`.

Différences avec la présentation de S. Baro La version de PAF! que nous présentons ici diffère de [2] sur plusieurs aspects liés notamment à la suppression du typage statique. Ainsi les variables de relations sont-elles indexées uniquement par leur arité (et non plus par leur type), tandis qu'une règle d'inférence telle que

$$\frac{\vdash \forall x \downarrow \mathtt{nat}.\ A(x) \qquad \vdash \mathtt{true} \downarrow \mathtt{nat}}{\vdash A(\mathtt{true})}\ \forall^1 e$$

devient correcte dans notre système, contrairement à la présentation de S. Baro (bien entendu, la prémisse $\vdash \mathtt{true} \downarrow \mathtt{nat}$ ne pourra pas être prouvée). De plus, l'utilisation d'une algèbre de types étendue fait apparaître de nouvelles règles dans le système logique, notamment les règles relatives aux quantifications des variables de types sous le prédicat de typage fort ($\forall\downarrow i$, $\forall\downarrow e$).

Comparaison avec d'autres systèmes Le système PAF! n'est pas conçu autour de la correspondance de Curry-Howard ; les programmes et les preuves vivent dans des mondes séparés. Dans Coq les preuves sont des programmes, et l'impératif de cohérence logique nécessite la mise en place de restrictions sévères pour forcer la normalisation de toutes les preuves, et donc la terminaison de tous les programmes (typiquement : les conditions de positivité dans les types inductifs et les conditions de garde dans les `Fixpoint`). Sur le plan théorique ces restrictions complexifient considérablement la construction des modèles de Coq et la preuve de sa cohérence. Sur un plan plus pratique, elles rendent difficile l'écriture de fonctions définies par récurrence non structurelle et plus encore l'écriture de fonctions partielles. Certaines tactiques comme `Function` permettent de faciliter en partie la tâche à l'utilisateur au prix de transformations très lourdes au niveau du formalisme sous-jacent. Au contraire, la séparation des preuves et des programmes à l'œuvre dans PAF! permet la gestion native de programmes divergents et/ou mal typés (ainsi que la preuve de certaines de leurs propriétés) sans mettre en danger la cohérence du formalisme ni la simplicité du modèle qui la justifie.

Une autre approche pour traiter les problèmes de récursion non structurelle, la *type based termination* [1], consiste à intégrer dans les types des indications de taille pour contrôler la décroissance des arguments lors des appels récursifs. Le système de types dépendants qui en résulte est décidable et garantit la terminaison d'une classe de fonctions récursives plus large que celle dont la terminaison est garantie par le seul critère de décroissance structurelle. Ces techniques (compatibles avec la correspondance de Curry-Howard) peuvent sans doute être incorporées à PAF ! [3] au prix d'une complexification notable du système.

2. Description formelle de PAF!

2.1. Le langage de programmation

Les termes Les termes du langage reprennent les constructions du fragment fonctionnel de ML qui permettent de manipuler les booléens, les entiers naturels, les listes et les fonctions. Formellement, on se donne un ensemble dénombrable $\mathbb{X}$ de variables de termes (notation : $x, y, z, \ldots$). Les termes (notation : $t, u, \ldots$) sont définis par la grammaire suivante :

$$\begin{array}{rcl} t & := & x \mid () \\ & \mid & \texttt{fun}\ x \to t \mid t\ t \mid \texttt{let}\ x = t\ \texttt{in}\ t \mid \texttt{fix}\ t \\ & \mid & \texttt{true} \mid \texttt{false} \mid t = t \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\ & \mid & 0 \mid \texttt{succ}(t) \mid (\texttt{match}_n\ t\ \texttt{with}\ \ 0 \to t \mid \texttt{succ}(x) \to t) \\ & \mid & \texttt{[]} \mid t :: t \mid (\texttt{match}_l\ t\ \texttt{with}\ \ \texttt{[]} \to t \mid x :: y \to t) \end{array}$$

L'ensemble des (occurrences des) variables libres d'un terme t est défini de la manière habituelle, il est noté $FV(t)$. L'ensemble des termes est noté $\mathbb{T}(\mathbb{X})$ et celui des termes clos $\mathbb{T}$. L'ensemble des valeurs, $\mathbb{V}(\mathbb{X})$, est un sous-ensemble de $\mathbb{T}$ défini par :

$$v := x \mid () \mid \texttt{true} \mid \texttt{false} \mid 0 \mid \texttt{succ}(v) \mid \texttt{[]} \mid v :: v \mid \texttt{fun}\ x \to t$$

Remarque : le langage est en appel par valeur, les variables désignent donc des valeurs et par extension (à travers la règle de conversion), des programmes s'évaluant sur des valeurs. De plus, les variables sont des valeurs afin de permettre l'évaluation symbolique des programmes(cf section 2.2).

La substitution sur les termes (notée $t\{x \mapsto u\}$) est définie de la manière habituelle. Soit $\sigma = [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ une liste finie d'associations variable/terme, $t[\sigma]$ est obtenu par *substitution parallèle* en remplaçant dans t chaque (occurrence d'une) variable libre x par $\sigma(x)$: seules les occurrences libres dans t sont substituées, et non les occurrences libres dans les t_i.

L'évaluation Il s'agit d'une évaluation de tête faible, en appel par valeur, dotée d'une stratégie à « petits pas » avec substitution. L'ordre d'évaluation (de droite à gauche comme en Caml) est fixé au moyen des *contextes*.

Définition 1 (Contexte)**.**

Les *contextes* d'évaluation sont définis par la syntaxe abstraite suivante :

$$\begin{array}{rcl} \mathbf{C}[\,] & ::= & \texttt{[]} \mid t\ \mathbf{C}[\,] \mid (\mathbf{C}[\,])v \mid \texttt{let}\ x = \mathbf{C}[\,]\ \texttt{in}\ u \mid \texttt{fix}\ \mathbf{C}[\,] \\ & \mid & t = \mathbf{C}[\,] \mid \mathbf{C}[\,] = v \mid \texttt{if}\ \mathbf{C}[\,]\ \texttt{then}\ t\ \texttt{else}\ u \\ & \mid & \texttt{succ}(\mathbf{C}[\,]) \mid (\texttt{match}_n\ \mathbf{C}[\,]\ \texttt{with}\ \ 0 \to u_1 \mid \texttt{succ}(x) \to u_2) \\ & \mid & t :: \mathbf{C}[\,] \mid \mathbf{C}[\,] :: v \mid (\texttt{match}_l\ \mathbf{C}[\,]\ \texttt{with}\ \ \texttt{[]} \to u_1 \mid x :: y \to u_2) \end{array}$$

Par définition, deux valeurs v et v' sont comparables si elles sont syntaxiquement identiques ou s'il s'agit de chaînes de constructeurs, en particulier : $\texttt{fun}\ x \to t = \texttt{fun}\ x \to t$ est défini, tandis que $\texttt{fun}\ x \to t = \texttt{fun}\ x \to t'$ ne l'est pas.

L'évaluation en une étape d'un terme t en un terme t', notée $t \ \rhd\ t'$, est définie à partir de l'*évaluation à la racine* (deux β-réductions et un axiome pour chaque opérateur), et par passage au contexte. La relation $\rightsquigarrow$ est la clôture réflexive et transitive de $\rhd$.

Définition 2 (Évaluation)**.**

$(\texttt{fun}\ x \to t)v \ \rhd\ t[x \mapsto v]$	(β-fun)
$\texttt{let}\ x = v\ \texttt{in}\ t \ \rhd\ t[x \mapsto v]$	(β-letin)
$v = v \ \rhd\ \texttt{true}$	($\rhd$égalité)
$v = v' \ \rhd\ \texttt{false}$ v et v' comparables et différentes	($\rhd$diff)

$$\texttt{if true then } t \texttt{ else } u \ \triangleright\ t \qquad (\triangleright\text{if-true})$$

$$\texttt{if false then } t \texttt{ else } u \ \triangleright\ u \qquad (\triangleright\text{if-false})$$

$$\texttt{fix (fun } f \to t) \ \triangleright\ t[f \mapsto \texttt{fix (fun } f \to t)] \qquad (\triangleright\texttt{fix})$$

$$\left(\begin{array}{l}\texttt{match}_n\ 0\ \ \texttt{with}\\ \qquad\qquad \mid\ 0 \to t_1\\ \qquad\qquad \mid\ \texttt{succ}(x) \to t_2\end{array}\right) \triangleright\ t_1 \qquad (\triangleright\texttt{matchnat})$$

$$\left(\begin{array}{l}\texttt{match}_n\ \texttt{succ}(t)\ \ \texttt{with}\\ \qquad\qquad \mid\ 0 \to t_1\\ \qquad\qquad \mid\ \texttt{succ}(x) \to t_2\end{array}\right) \triangleright\ t_2[x \mapsto t] \qquad (\triangleright\texttt{matchnat})$$

$$\left(\begin{array}{l}\texttt{match}_l\ \texttt{[]}\ \ \texttt{with}\\ \qquad\qquad \mid\ \texttt{[]} \to t_1\\ \qquad\qquad \mid\ x :: y \to t_2\end{array}\right) \triangleright\ t_1 \qquad (\triangleright\texttt{matchlist})$$

$$\left(\begin{array}{l}\texttt{match}_l\ t :: u\ \ \texttt{with}\\ \qquad\qquad \mid\ \texttt{[]} \to t_1\\ \qquad\qquad \mid\ x :: y \to t_2\end{array}\right) \triangleright\ t_2[x \mapsto t; y \mapsto u] \qquad (\triangleright\texttt{matchlist})$$

$$\frac{t \ \triangleright\ t'}{\mathbf{C}[t] \ \triangleright\ \mathbf{C}[t']} \qquad (\text{contexte})$$

En l'absence de typage, l'évaluation de t peut ne pas terminer ($t\uparrow$), terminer sur une valeur ($t \rightsquigarrow v$) ou sur un terme qui n'est pas une valeur ($t \rightsquigarrow t' \not\rightsquigarrow$).

2.2. Le langage de spécification

Les types L'algèbre de types est formée à partir d'un ensemble dénombrable, $\mathcal{A}$, de variables de types $(\alpha, \beta, \ldots)$, ainsi que des types constants pour les booléens, les entiers naturels et les listes. Elle est construite inductivement à l'aide de la flèche et de la quantification universelle :

$$\tau \ ::= \ \alpha \mid \texttt{unit} \mid \texttt{nat} \mid \texttt{bool} \mid \tau\ \texttt{list} \mid \tau \to \tau \mid \forall\alpha.\tau$$

$\mathcal{T}(\mathcal{A})$ dénote l'ensemble des types ouverts, tandis que $\mathcal{T}$ dénote l'ensemble des types clos. Les variables libres et liées, ainsi que la substitution sont ici aussi définies de manière standard.

Les formules Le langage de spécification est une logique multi-sortée qui étend le langage de l'arithmétique du second ordre. On y retrouve la sorte des individus (ici, les valeurs) ainsi qu'une sorte des relations n-aires sur les individus pour chaque arité $n \geq 0$, plus une sorte des types ML.

On notera que le langage des formules ne comporte pas de symbole d'égalité ni les constantes propositionnelles $\bot$ et $\top$. Le symbole de prédicat I permet en effet de faire remonter au niveau des formules l'égalité booléenne du langage de programmation sous-jacent, et de représenter les formules $\top$ et $\bot$ à l'aide des abréviations $\top = I(true)$ et $\bot = I(false)$.

Les variables du premier ordre sont ici confondues avec les variables du langage fonctionnel sous-jacent ; elles représentent donc des valeurs. Par extension, on autorisera la substitution de ces variables par des termes quelconques (règles $\forall^1 e$ et $\exists^1 i$), sous réserve que ceux-ci se réduisent sur des valeurs. Techniquement, cette restriction est obtenue en ajoutant aux prémisses des règles ($\forall^1 i$ et $\exists^1 e$) une condition de typage fort qui garantit que le terme substitué se réduit effectivement sur une valeur.

Contrairement à la présentation donnée dans [2], la quantification universelle du second ordre est indexée uniquement par l'arité des variables de relation (notation : X^n pour désigner une variable du 2nd ordre d'arité n).

Formellement, l'ensemble $\mathscr{F}$ des formules $(A, B, C, \ldots)$ est défini inductivement par la grammaire suivante, construite à partir d'un ensemble dénombrable, $\mathscr{X}$, de variables de relation $(X, Y, Z, \ldots)$:

$$\begin{array}{rcl} A & ::= & X(t,\ldots,t) \mid I(t) \mid t \downarrow \tau \\ & & \mid \neg A \mid A \to A \mid A \wedge A \mid A \vee A \\ & & \mid \forall x{\downarrow}\tau.\ A \mid \exists x{\downarrow}\tau.\ A \mid \forall \alpha.\ A \mid \forall X^n.\ A \end{array}$$

Une formule A peut contenir comme variables libres aussi bien des variables de terme (celles qui apparaissent libres dans un terme de A), que des variables de type (libres dans un type de A) ou des variables de relation. Soit $t_1,\ldots,t_n$ des termes, $x_1,\ldots,x_n$ des variables de terme, et A une formule. On suppose que les x_i et les variables libres des t_i ne sont pas liées dans A (toujours possible par α-conversion). La formule $A[x_1 \mapsto t_1,\ldots,x_n \mapsto t_n]$ est obtenue en remplaçant dans A chaque occurrence libre de x_i par t_i ($1 \leq i \leq n$). L'ensemble des variables de relations libres d'une formule A, noté $FV(A)$ est défini de la même manière que pour les termes ou les types. La formule $A[X(x_1,\ldots,x_n) \mapsto B]$, où A et B sont deux formules, X est une variable de relation n-aire et $x_1,\ldots,x_n$ sont des variables de terme, dénote (cf [5]) la substitution de B à $X(x_1,\ldots,x_n)$ dans A [2]. La définition se fait par induction sur A (dans laquelle $x_1,\ldots,x_n$ et les variables libres de B ne sont pas liées) :

Définition 3 (Substitution du second ordre)**.**

$$\begin{array}{lcl} X(t_1,\ldots,t_n)[X(x_1,\ldots,x_n) \mapsto B] & = & B[x_1 \mapsto t_1,\ldots,x_n \mapsto t_n] \\ I(t)[X(x_1,\ldots,x_n) \mapsto B] & = & I(t) \\ (t \downarrow \tau)[X(x_1,\ldots,x_n) \mapsto B] & = & t \downarrow \tau \\ (A_1\, c\, A_2)[X(x_1,\ldots,x_n) \mapsto B] & = & A_1[X(x_1,\ldots,x_n) \mapsto B]\, c\, A_n[X(x_1,\ldots,x_n) \mapsto B] \end{array}$$

où c est un connecteur. Le cas de la négation est similaire.

$$(\mathtt{Q}x{\downarrow}\tau.\ A)[X(x_1,\ldots,x_n) \mapsto B] \quad = \quad \mathtt{Q}x{\downarrow}\tau.\ A[X(x_1,\ldots,x_n) \mapsto B]$$

où $\mathtt{Q}$ peut être l'une des deux quantifications du premier ordre.

$$(\forall \xi.\ A)[X(x_1,\ldots,x_n) \mapsto B] \quad = \quad \forall \xi.\ A[X(x_1,\ldots,x_n) \mapsto B]$$

ξ étant soit une variable de type, soit une variable de relation différente de X.

$$(\forall X^n.\ A)[X(x_1,\ldots,x_n) \mapsto B] \quad = \quad \forall X^n.\ A$$

La logique Nous en venons maintenant aux règles qui constituent le système déductif de PAF! Celles-ci sont exposées dans le formalisme de la déduction naturelle sous forme de séquents ($\Gamma \vdash A$: la formule A est prouvable à partir de Γ, un ensemble fini de formules, $FV(\Gamma)$ dénote l'ensemble des variables qui apparaissent libres dans l'une au moins des formules de Γ). Ce système contenant un assez grand nombre de règles, nous allons les introduire par petits groupes. Tout d'abord, axiomes et règles pour les constantes $\top = I(\mathtt{true})$ et $\bot = I(\mathtt{false})$:

$$\frac{}{\Gamma, A \vdash A}\ \text{ID} \qquad \frac{}{\Gamma \vdash \top}\ \text{TOP} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \text{EX-FALSO}$$

Ensuite les règles logiques dont le fragment propositionnel est standard :

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg i \qquad \frac{\Gamma \vdash A}{\Gamma, \neg A \vdash \bot}\ \neg e$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge i \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge e\text{G} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge e\text{D}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee i\text{G} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee i\text{D} \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee e$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to i \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash A \to B}{\Gamma \vdash B}\ \to e$$

2. On aurait pu choisir une définition équivalente : $A[X \mapsto \lambda x_1 \ldots x_n.B]$ où le symbole X est remplacé par la fonction propositionnel $\lambda x_1 \ldots x_n.B$.

Les règles d'introduction et d'élimination des quantifications bornées du 1er ordre sont les règles habituelles que l'on a modifiées pour tenir compte des contraintes de typage fort. Ces modifications se comprennent aisément à travers la décomposition standard [4] des quantificateurs bornés ($\forall x{\downarrow}\tau.\ A \ \equiv\ \forall x.(x \downarrow \tau \rightarrow A)$ et $\exists x{\downarrow}\tau.\ A \ \equiv\ \exists x.(x \downarrow \tau \wedge A)$) à partir des quantificateurs non bornés qui ne figurent pas dans notre langage. Les autres règles sont habituelles [5] :

$$\frac{\Gamma,\ x \downarrow \tau \vdash A}{\Gamma \vdash \forall x{\downarrow}\tau.\ A}\ {\scriptstyle x \notin FV(\Gamma)}\ \forall^1 i \qquad \frac{\Gamma \vdash \forall x{\downarrow}\tau.\ A \qquad \Gamma \vdash t \downarrow \tau}{\Gamma \vdash A[x \mapsto t]}\ \forall^1 e$$

$$\frac{\Gamma \vdash A[x \mapsto t] \qquad \Gamma \vdash t \downarrow \tau}{\Gamma \vdash \exists x{\downarrow}\tau.\ A}\ \exists i \qquad {\scriptstyle x \notin FV(\Gamma,C)}\ \frac{\Gamma \vdash \exists x{\downarrow}\tau.\ A \qquad \Gamma,\ x \downarrow \tau,\ A \vdash C}{\Gamma \vdash C}\ \exists e$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall \alpha.\ A}\ {\scriptstyle \alpha \notin FV(\Gamma)}\ \forall^\alpha i \qquad \frac{\Gamma \vdash \forall \alpha.\ A}{\Gamma \vdash A[\alpha \mapsto \tau]}\ \forall^\alpha e$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall X^n.\ A}\ {\scriptstyle X \notin FV(\Gamma)}\ \forall^2 i \qquad \frac{\Gamma \vdash \forall X^n.\ A}{\Gamma \vdash A[X x_1 ... x_n \mapsto B]}\ \forall^2 e$$

Le système comporte également des règles pour raisonner sur les types de base :

$$\frac{\Gamma \vdash A[x \mapsto \texttt{true}] \qquad \Gamma \vdash A[x \mapsto \texttt{false}]}{\Gamma \vdash \forall x{\downarrow}\texttt{bool}.\ A}\ \textsc{Case-bool}$$

$$\frac{\Gamma \vdash A[x \mapsto 0] \qquad \Gamma,\ n \downarrow \texttt{nat} \vdash A[x \mapsto n] \rightarrow A[x \mapsto \texttt{succ}(n)]}{\Gamma \vdash \forall x{\downarrow}\texttt{nat}.\ A}\ {\scriptstyle n \notin FV(\Gamma)}\ \textsc{rec-nat}$$

$$\frac{\Gamma \vdash A[x \mapsto []] \qquad \Gamma,\ t \downarrow \tau,\ l \downarrow \tau\ \texttt{list} \vdash A[x \mapsto l] \rightarrow A[x \mapsto t :: l]}{\Gamma \vdash \forall x{\downarrow}\tau\ \texttt{list}.\ A}\ {\scriptstyle n \notin FV(\Gamma)}\ \textsc{rec-}\ \texttt{list}$$

Bien entendu, le système est doté de règles relatives au typage fort qui vont servir à montrer (en partie) la terminaison des programmes :

$$\frac{\Gamma \vdash t \downarrow \tau}{\Gamma \vdash t \downarrow \forall \alpha.\ \tau}\ {\scriptstyle \alpha \notin FV(\Gamma)}\ \forall{\downarrow}\, i \qquad \frac{\Gamma \vdash t \downarrow \forall \alpha.\ \tau}{\Gamma \vdash t \downarrow \tau[\alpha \mapsto \tau']}\ \forall{\downarrow}\, e$$

$$\frac{}{\Gamma \vdash () \downarrow \texttt{unit}}\ \textsc{X-unit} \qquad \frac{}{\Gamma \vdash \texttt{true} \downarrow \texttt{bool}}\ \textsc{X-bool T} \qquad \frac{}{\Gamma \vdash \texttt{false} \downarrow \texttt{bool}}\ \textsc{X-bool F}$$

$$\frac{}{\Gamma \vdash 0 \downarrow \texttt{nat}}\ \textsc{X-0} \qquad \frac{\Gamma \vdash t \downarrow \texttt{nat}}{\Gamma \vdash \texttt{succ}(t) \downarrow \texttt{nat}}\ \textsc{X-nat}$$

$$\frac{}{\Gamma \vdash \texttt{[]} \downarrow \forall \alpha.\ (\alpha\ \texttt{list})}\ \textsc{X-[]} \qquad \frac{\Gamma \vdash t \downarrow \tau \qquad \Gamma \vdash l \downarrow \tau\ \texttt{list}}{\Gamma \vdash t :: l \downarrow \tau\ \texttt{list}}\ \textsc{X-}\tau\ \textsc{list}$$

$$\frac{\Gamma \vdash t \downarrow \tau \rightarrow \tau' \qquad \Gamma \vdash u \downarrow \tau}{\Gamma \vdash (t)u \downarrow \tau'}\ \textsc{X-app} \qquad \frac{\Gamma, x \downarrow \tau \vdash t \downarrow \tau'}{\Gamma \vdash \texttt{fun}\ x \rightarrow t \downarrow \tau \rightarrow \tau'}\ \textsc{X-fun}$$

Ces règles ne mentionnent cependant pas toutes les constructions du langage. Celles-ci sont gérées par les règles d'évaluation, cruciales dans le système et qui font appel à un évaluateur intégré (évaluation symbolique). Elles permettent d'identifier tout programme à sa valeur, lorsqu'il en a une, et expriment l'invariance des propriétés vis-à-vis de l'évaluation : le langage de spécification ne permet pas de distinguer deux termes β-équivalents.

$$\frac{\Gamma \vdash A[x \mapsto t'] \qquad t \rightsquigarrow t'}{\Gamma \vdash A[x \mapsto t]}\ \textsc{eval G} \qquad \frac{\Gamma \vdash A[x \mapsto t] \qquad t \rightsquigarrow t'}{\Gamma \vdash A[x \mapsto t']}\ \textsc{eval D}$$

À cause de la seconde prémisse et de la présence de termes divergents, vérifier une instance de ces règles (et *a fortiori* une preuve dans PAF!) n'est pas décidable. En pratique, il est possible de forcer la décidabilité de la vérification des règles d'évaluation en les restreignant, par exemple en limitant le nombre d'étapes d'évaluation à un nombre fixé à l'avance (éventuellement paramétrable par l'utilisateur). On ne considérera pas cette restriction ici, qui est davantage un problème de design pour l'implémentation.

Définition 4.

Une formule A est *prouvable* si le séquent $\vdash A$ peut être dérivé à l'aide des règles ci-dessus. Un terme t est *fortement de type* τ si la formule $t \downarrow \tau$ est prouvable.

Exemples : On donne ci-dessous une preuve de la totalité de la fonction factorielle : $\forall n \downarrow \texttt{nat}.\ \texttt{fact}\ n \downarrow \texttt{nat}$. On suppose donnée une preuve de typage fort de la multiplication et on dérive :

$$\dfrac{\dfrac{}{\vdash * \downarrow \texttt{nat} \to \texttt{nat} \to \texttt{nat}}\,\textsc{thm} \qquad \dfrac{\dfrac{}{n \downarrow \texttt{nat} \vdash n \downarrow \texttt{nat}}\,\textsc{id}}{n \downarrow \texttt{nat} \vdash \texttt{succ}(n) \downarrow \texttt{nat}}\,\textsc{xnat} \qquad \dfrac{}{\texttt{fact}\ n \downarrow \texttt{nat} \vdash \texttt{fact}\ n \downarrow \texttt{nat}}\,\textsc{id}}{S: \quad n \downarrow \texttt{nat}, \texttt{fact}\ n \downarrow \texttt{nat} \vdash \texttt{succ}(n) * (\texttt{fact}\ n) \downarrow \texttt{nat}}\,\textsc{xapp}$$

On peut maintenant construire la dérivation :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash 0 \downarrow \texttt{nat}}\,\textsc{x0}}{\vdash 1 \downarrow \texttt{nat}}\,\textsc{xnat} \qquad \texttt{fact}\ 0 \rightsquigarrow 1}{\vdash \texttt{fact}\ 0 \downarrow \texttt{nat}}\,\textsc{evg} \qquad \dfrac{S \qquad \texttt{fact}\ (\texttt{succ}(n)) \rightsquigarrow \texttt{succ}(n) * (\texttt{fact}\ n)}{n \downarrow \texttt{nat}, \texttt{fact}\ n \downarrow \texttt{nat} \vdash \texttt{fact}\ (\texttt{succ}(n)) \downarrow \texttt{nat}}\,\textsc{evg}}{\vdash \forall n \downarrow \texttt{nat}.\ \texttt{fact}\ n \downarrow \texttt{nat}}\,\textsc{rec nat}$$

On peut prouver que la fonction `let rec div` n `= if (`n `mod 2) = 0 then` $n/2$ `else div` n est définie sur tous les entiers pairs (on s'autorise l'usage d'une tactique de réécriture (RW), présente dans l'implémentation actuelle) :

$$\dfrac{\dfrac{\exists x \downarrow \texttt{nat}.\ I(n = 2x) \ \vdash\ \exists x \downarrow \texttt{nat}.\ I(n = 2x) \qquad \dfrac{\dfrac{\dfrac{}{n \downarrow \texttt{nat}, x \downarrow \texttt{nat}, I(n = 2x) \ \vdash\ x \downarrow \texttt{nat}}\,\textsc{id} \qquad \texttt{div}\ 2x \rightsquigarrow x}{n \downarrow \texttt{nat}, x \downarrow \texttt{nat}, I(n = 2x) \ \vdash\ \texttt{div}\ 2x \downarrow \texttt{nat}}\,\textsc{evalg}}{n \downarrow \texttt{nat}, x \downarrow \texttt{nat}, I(n = 2x) \ \vdash\ \texttt{div}\ n \downarrow \texttt{nat}}\,\textsc{rw}}{n \downarrow \texttt{nat},\ \exists x \downarrow \texttt{nat}.\ I(n = 2x) \ \vdash\ \texttt{div}\ n \downarrow \texttt{nat}}\,\exists^1 i}{\vdash\ \forall n \downarrow \texttt{nat}.\ ((\exists x \downarrow \texttt{nat}.\ I(n = 2x)) \to \texttt{div}\ n \downarrow \texttt{nat})}\,\forall^1 i\ +\ \to i$$

Remarque : Le lecteur aura remarqué que chaque assertion de typage fort nécessite une preuve, ce qui peut vite devenir fastidieux pour l'utilisateur. Il est cependant possible d'automatiser une partie de ce travail en remarquant qu'une dérivation de typage du système F (qui garantit la terminaison du programme) peut être automatiquement transformée en une preuve de typage fort à l'aide du lemme suivant :

Lemme 1.

Si $x_1 : \tau_1, ..., x_n : \tau_n \vdash t : \tau$ est un jugement de typage dérivable dans le système F, alors le séquent $x_1 \downarrow \tau_1, \ldots, x_n \downarrow \tau_n \vdash t \downarrow \tau$ est dérivable dans PAF!

Démonstration :

La preuve se fait par induction sur la structure de la dérivation, en utilisant les règles ID, X-FUN, X-APP, $\forall^\alpha i$ et $\forall^\alpha e$ pour mimer les règles de typage du système F. La preuve s'adapte facilement au système F étendu avec un let-in, les entiers et les booléens (mais sans fix) à l'aide de l'évaluation symbolique. □

En pratique, on pourra donc automatiser la preuve de typage fort pour tout fragment du système F dans lequel la vérification de type est décidable, comme par exemple le système de types de ML restreint aux termes sans fix.

3. Le modèle

Cette section est consacrée à la construction d'un modèle de PAF! Nous attendons de ce modèle, qu'il garantisse que la théorie du typage fort est adéquate, c'est-à-dire qu'elle capture bien la notion de terminaison, et cela alors même que le typage statique a été supprimé. On attend aussi qu'il garantisse la cohérence de la logique utilisée par le moteur de preuve.

Nous reprenons le modèle présenté dans [3], auquel nous ajoutons les booléens, les listes, ainsi que les types universels imprédicatifs du système F. De plus, nous étendons ce modèle, initialement dédié à la seule correction du typage fort, à tout le système de spécification permettant ainsi de montrer la cohérence de la logique.

Il s'agit d'un modèle simple de réalisabilité dans lequel les termes sont interprétés par des termes clos et les types par des ensembles, clos par β-réduction et β-expansion, de termes eux-mêmes clos, chacun se réduisant sur une valeur. Ces ensembles seront appelés des *candidats*. Les formules sont quant à elles interprétées par des valeurs de vérité. Le fait de considérer des types quantifiés, nécessite de tenir compte des variables de types auxquelles une interprétation doit être donnée. De même, il est nécessaire de donner une interprétation aux variables de relation. Pour cette raison, nous avons besoin des *valuations* dont le but est de prendre en charge les opérations de clôture.

3.1. Définitions

Les candidats L'ensemble de tous les candidats est noté $\mathbb{C}$. Soit $V \subset \mathbb{V}$, un ensemble quelconque de valeurs, la clôture par β-expansion de V sera, de manière standard, notée $\uparrow V$.

Définition 5.

Un *candidat* C est un ensemble de termes clos tel que :
- C est clos à la fois par β-réduction et par β-expansion
- si $t \in C$, alors t se réduit sur une valeur

La caractérisation suivante fournit un critère simple pour déterminer si un ensemble donné de termes est ou non un candidat et elle fournit également une méthode pour construire un candidat :

Caractérisation : Les trois propositions suivantes sont équivalentes :
- C est un candidat
- C est l'expansion par β-réduction d'un ensemble quelconque de valeurs
- $C = \uparrow (C \cap \mathbb{V})$

La dernière proposition indique qu'un ensemble de programmes clos est un candidat, s'il est égal à la β-expansion de l'ensemble des valeurs qu'il contient. La seconde proposition nous assure que l'on peut toujours construire un candidat en prenant l'expansion d'un ensemble quelconque de valeurs. On peut donc caractériser l'ensemble des candidats par : $\mathbb{C} = \{\uparrow V \mid V \subset \mathbb{V}\}$. Distinguons deux éléments particulier de $\mathbb{C}$: $\emptyset$, l'élément minimal et $\uparrow\mathbb{V}$ l'élément maximal.

Remarque : Comme tout langage un tant soit peu expressif, l'algèbre de types est incomplète et ses modèles contiennent plus d'objets que le langage ne peut en définir. Aussi, il n'est pas surprenant de trouver dans $\mathbb{C}$ des candidats tels que l'ensemble $\uparrow\{0, \texttt{false}\}$ qui ne correspond à aucun type du langage.

Les fonctions propositionnelles Elles vont permettre d'interpréter les variables de relation :

Définition 6 (Fonctions propositionnelles)**.**

Une fonction f est β-compatible si pour tous termes $t_1, t'_1, \ldots, t_n, t'_n$ tels que $t_i \sim_\beta t'_i$ $(1 \leq i \leq n)$, $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$. L'ensemble $\mathcal{F}^n_{\mathcal{P}}$ des fonctions propositionnelles d'arité n est défini ci-

dessous, et l'ensemble des fonctions propositionnelles est la réunion des $\mathcal{F}_{\mathcal{P}}^n$ pour tous les entiers n :

$$\begin{array}{rcl} \mathcal{F}_{\mathcal{P}}^n & = & \{f : \mathbb{T}^n \longrightarrow \{0,1\} \mid f \text{ est } \beta\text{-compatible}\} \\ \mathcal{F}_{\mathcal{P}} & = & \bigcup_{n\in\mathbb{N}} \mathcal{F}_{\mathcal{P}}^n \end{array}$$

Les valuations Le rôle des valuations est de clore les termes, les types et les formules. Elles sont donc définies à la fois sur $\mathbb{X}$ (variables de termes), $\mathcal{A}$ (variables de types) et $\mathscr{X}$ (variables de relation). Une *valuation* (notée ρ) est une liste d'associations où chaque variable de terme est associée à un terme clos, chaque variable de type à un candidat et chaque variable de relation n-aire à une fonction propositionnelle n-aire ($\rho(x) \in \mathbb{T}$, $\rho(\alpha) \in \mathbb{C}$ et $\rho(X^n) \in \mathcal{F}_{\mathcal{P}}^n$). Bien que les valuations soient définies sur les variables des trois catégories syntaxiques, l'interprétation (dans une valuation) des termes ne dépend que de la valuation restreinte aux variables de termes et celle des types ne dépend que de la valuation restreinte aux variables de types. Par contre, l'interprétation des formules dépend de la valuation sur toutes les catégories.

Remarque : étant donnée une valuation ρ, la restriction de ρ aux variables de termes est une liste d'association (variable de terme)-(terme clos) et on peut en extraire une substitution. Par abus de langage, on notera $t[\rho]$ le terme obtenu par substitution parallèle où l'on considère exclusivement la restriction de ρ aux variables de terme.

Définition 7 (Les valuations)**.**

Soit ρ une valuation. Alors $\rho; i \mapsto o$ est la valuation qui à i' associe $\rho(i')$ si $i' \neq i$ et o sinon. Deux valuations ρ et ρ' sont équivalentes si à toutes variables de termes x elles associent deux termes β-équivalents : $\rho \sim_\beta \rho' \quad \equiv \quad \forall x \in \mathbb{X}\ \rho(x) \sim_\beta \rho'(x)$

3.2. L'interprétation

L'interprétation d'un terme t (resp. d'un type τ) dans une valuation ρ est notée $[\![t]\!]_\rho$ (resp. $[\![\tau]\!]_\rho$). On pourra parfois omettre la valuation, notamment lorsque le terme (resp. le type) interprété est clos. L'interprétation des termes se définit trivialement au moyen de la substitution parallèle : $[\![t]\!]_\rho = t[\rho]$. On vérifie aisément que l'interprétation d'un terme est un programme clos, et que si t est un terme clos, alors $[\![t]\!]_\rho = t$. On en déduit sans difficulté, par simple induction sur les termes, la proposition suivante.

Proposition 1.

Soit t, t', deux termes, x une variable de terme libre dans t et ρ une valuation quelconque. Alors, $[\![t[x \mapsto t']]\!]_\rho = [\![t]\!]_{\rho; x\mapsto [\![t']\!]_\rho}$.

L'interprétation des types L'interprétation des types dans une valuation donnée ρ est définie par induction (sur les types). On notera $\mathtt{succ}^n(0)$ pour désigner $\underbrace{\mathtt{succ}(\ldots\mathtt{succ}(0))}_{\times n}$.

Définition 8 (Interprétation des types)**.**

$$\begin{array}{rcl|rcl} [\![\alpha]\!]_\rho & = & \rho(\alpha) & [\![\tau\ \mathtt{list}]\!]_\rho & = & \{t \in \mathbb{T} \mid \exists n \in \mathbb{N} \exists v_1,\ldots,v_n \in [\![\tau]\!]_\rho \cap \mathbb{V}\ t \rightsquigarrow [v_1,\ldots,v_n]\} \\ [\![\mathtt{bool}]\!]_\rho & = & \uparrow\{\mathtt{true}, \mathtt{false}\} & [\![\tau \to \tau']\!]_\rho & = & \{t \in \uparrow\mathbb{V} \mid \forall u \in [\![\tau]\!]_\rho \quad (t)u \in [\![\tau']\!]_\rho\} \\ [\![\mathtt{nat}]\!]_\rho & = & \bigcup_{n\in\mathbb{N}} \uparrow\{\mathtt{succ}^n(0)\} & [\![\forall\alpha.\tau]\!]_\rho & = & \bigcap_{e\in\mathbb{C}} ([\![\tau]\!]_{\rho;\alpha\mapsto e}) \end{array}$$

L'interprétation des types est standard, sauf en ce qui concerne la flèche qui est interprétée à l'aide de la flèche de la réalisabilité [3]. Toutefois, on ajoute une restriction supplémentaire : supposons que les

3. La flèche de la réalisabilité (notée $\Rightarrow$)est définie par : $A \Rightarrow B = \{e \mid \forall a \in A\ (e)a \in B\}$

types fonctionnels soient interprétés exactement par la flèche de la réalisabilité. Dans ce cas, si $[\![\tau]\!]_\rho$ est vide alors $[\![\tau \to \tau']\!]_\rho$ est $\mathbb{T}$ tout entier : tous les termes de $\mathbb{T}$ satisfont trivialement la condition, puisque la prémisse de l'implication est toujours fausse. Afin d'éviter ce problème, on restreint l'interprétation d'un type fonctionnel au candidat maximal $\uparrow\mathbb{V}$, autrement dit aux programmes qui terminent.

Proposition 2.

Soit τ, τ', deux types, α une variable de type libre dans τ et ρ une valuation quelconque. Alors, $[\![\tau[\alpha \mapsto \tau']]\!]_\rho = [\![\tau]\!]_{\rho;\alpha\mapsto[\![\tau']\!]_\rho}$.

Proposition 3.

Quel que soit le type τ et la valuation ρ, l'interprétation de τ dans ρ est un candidat : $[\![\tau]\!]_\rho \in \mathbb{C}$.

Remarque : les candidats sont construits à partir de $\mathbb{V}$ et peuvent donc contenir des fonctions telles que `boucle` : `fun` $x \to$ (`fix` (`fun` $y \to$ `fun` $x \to (y)x$))x dont l'application à n'importe quel argument diverge. On pourrait penser que cela pose problème, mais ce n'est pas le cas : le danger en l'occurrence, serait que `boucle` soit dans (l'interprétation d') un type clos fonctionnel et puisse potentiellement être appliquée. Or, quelle que soit la valuation ρ, les seuls types fonctionnels auxquels `boucle` peut appartenir sont de la forme $[\![\alpha \to \tau]\!]_{\rho;\alpha\mapsto\emptyset}$ (soit c un candidat non vide et $v \in c$, (`boucle` v), qui n'admet pas de valeur, n'appartient pas à $[\![\tau]\!]_\rho$ et donc `boucle` $\notin [\![\alpha \to \tau]\!]_{\rho;\alpha\mapsto e}$), et garantissent que, tant que `boucle` apparaît dans un contexte où elle ne sera pas appliquée, aucune erreur, aucune évaluation infinie ne sera produite. Puisque `boucle` ne peut appartenir qu'à cette sorte de type, elle donc est inoffensive.

Exemples

- Il est facile de voir que le terme `succ` appartient à $[\![\texttt{nat} \to \texttt{nat}]\!]_\rho$.
- L'interprétation du type vide est l'ensemble vide : $[\![\forall\alpha.\ \alpha]\!]_\rho = \bigcap_{e\in\mathbb{C}} [\![\alpha]\!]_{\rho;\alpha\mapsto e} = \bigcap_{e\in\mathbb{C}} e = \emptyset$
- On peut aussi voir que l'égalité du langage n'est pas un terme fortement de type $\forall\alpha.\ (\alpha \to \alpha \to \texttt{bool})$. En effet, l'égalité du langage n'est pas toujours définie sur les fonctions : si $t \neq t'$, alors `fun` $x \to t =$ `fun` $x \to t'$ n'est pas défini et donc n'appartient pas à $[\![\texttt{bool}]\!]_\rho$. Par conséquent, ' = ' $\notin [\![\forall\alpha.\ (\alpha \to \alpha \to \texttt{bool})]\!]_\rho$.
- Certains candidats ne correspondent à aucun type du langage, mais sont tout de même « habités » : `let f` $x =$ `if` x `then` 0 `else` x `in f` appartient au candidat $[\![\texttt{bool} \to \alpha]\!]_{\alpha\mapsto\uparrow\{0,\texttt{false}\}}$.

L'interprétation des formules L'interprétation d'une formule A se définie sans difficulté et de manière standard par induction sur la structure de A :

Définition 9 (Interprétation des formules)**.**

$$
\begin{aligned}
[\![X(t_1,\ldots,t_n)]\!]_\rho &= \rho(X)([\![t_1]\!]_\rho,\ldots,[\![t_n]\!]_\rho)\\
[\![I(t)]\!]_\rho &= \begin{cases} 1 & \text{si } [\![t]\!]_\rho \sim_\beta \texttt{true}\\ 0 & \text{si } [\![t]\!]_\rho \sim_\beta \texttt{false}\end{cases}\\
[\![\neg A]\!]_\rho &= 1 - [\![A]\!]_\rho\\
[\![A \wedge B]\!]_\rho &= \inf([\![A]\!]_\rho, [\![B]\!]_\rho)\\
[\![A \vee B]\!]_\rho &= \sup([\![A]\!]_\rho, [\![B]\!]_\rho)\\
[\![A \to B]\!]_\rho &= \sup(1 - [\![A]\!]_\rho, [\![B]\!]_\rho)
\end{aligned}
$$

$$
\begin{aligned}
[\![\forall x{\downarrow}\tau.\ A]\!]_\rho &= \inf_{v\in[\![\tau]\!]_\rho} ([\![A]\!]_{\rho;x\mapsto v})\\
[\![\exists x{\downarrow}\tau.\ A]\!]_\rho &= \sup_{v\in[\![\tau]\!]_\rho} ([\![A]\!]_{\rho;x\mapsto v})\\
[\![\forall\alpha.\ A]\!]_\rho &= \inf_{e\in\mathbb{C}} ([\![A]\!]_{\rho;\alpha\mapsto e})\\
[\![\forall X^n.\ A]\!]_\rho &= \inf_{f\in\mathcal{F}^n_{\mathcal{P}}} ([\![A]\!]_{\rho;X\mapsto f})\\
[\![t \downarrow \tau]\!]_\rho &= \begin{cases} 1 & \text{si } [\![t]\!]_\rho \in [\![\tau]\!]_\rho\\ 0 & \text{sinon}\end{cases}
\end{aligned}
$$

L'interprétation est également définie sur les ensembles de formules : $[\![\Gamma]\!]_\rho = \inf_{A\in\Gamma}([\![A]\!]_\rho)$. On pourra donc écrire $[\![\Gamma \to A]\!]_\rho$, et si cette expression vaut 1, alors $[\![\Gamma]\!]_\rho = 1$ implique $[\![A]\!]_\rho = 1$. On calcule aisément que $[\![I(\texttt{false})]\!]_\rho = 0$.

3.3. Correction et cohérence

Le but est de montrer que les règles du système ne permettent pas de dériver une contradiction, en l'occurrence $I(\texttt{false})$. Pour cela, nous allons montrer la correction de la logique vis-à-vis de notre modèle. De cette propriété de correction se déduira immédiatement la cohérence du système déductif. La démonstration fait intervenir plusieurs résultats intermédiaires que nous donnons ci-dessous.

Soit la formule A et la formule B, le terme t, le type τ, les variables $x, x_1, \ldots, x_n \in \mathbb{X}$, $\alpha \in \mathcal{A}$, $X \in \mathscr{X}$ et les valuations ρ et ρ', alors :

Lemme 2.

1. $[\![A[x \mapsto t]]\!]_\rho = [\![A]\!]_{\rho;x\mapsto[\![t]\!]_\rho}$.
2. $[\![A[\alpha \mapsto \tau]]\!]_\rho = [\![A]\!]_{\rho;\alpha\mapsto[\![\tau]\!]_\rho}$.
3. $\rho \sim_\beta \rho' \Longrightarrow [\![A]\!]_\rho = [\![A]\!]_{\rho'}$
4. $[\![A[Xx_1 \ldots x_n \mapsto B]]\!]_\rho = [\![A]\!]_{\rho;X\mapsto\lambda v_1\ldots v_n.\,[\![B]\!]_{\rho;x_1\mapsto v_1;\ldots;x_n\mapsto v_n}}$.

Remarque : le Lemme 2.3 entraîne que pour toute formule A et toute valuation ρ, la fonction $\lambda v_1, \ldots, v_n.\,[\![A]\!]_{\rho;x_1\mapsto v_1;\ldots;x_n\mapsto v_n}$ est une fonction propositionnelle β-compatible.

Théorème 1.

Quels que soient la formule A et le contexte Γ, si le séquent $\Gamma \vdash A$ est dérivable, alors pour toute valuation ρ, $[\![\Gamma \longrightarrow A]\!]_\rho = 1$.

La preuve du théorème se fait par induction sur la taille de la dérivation π du séquent $\Gamma \vdash A$. On montre, pour chaque règle que, si $[\![\Gamma' \to A']\!]_\rho = 1$ pour tout séquent prémisse $\Gamma' \vdash A'$, alors $[\![\Gamma \to A]\!]_\rho = 1$. Les règles pour les quantifications du premier ordre ont besoin du Lemme 1. Il garantit que la substitution d'un terme t à une variable x dans une formule revient à ajouter l'association $x \mapsto [\![t]\!]_\rho$ à la valuation. Le Lemme 2 assure un résultat similaire pour la substitution des types et intervient pour la quantification des types. Les règles pour les quantifications du second ordre quant à elles utilisent le Lemme 4. Enfin le cas des règles d'évaluation fait appel aux Lemmes 1 et 3, ce dernier garantissant l'invariance de l'interprétation d'une formule par rapport à la β-équivalence des valuations. En faisant la preuve, on pourra constater qu'il n'y est jamais fait mention du typage statique.

La correction du système déductif est un corollaire du théorème 1 dans le cas où $\Gamma = \emptyset$.

Corollaire 1 (Correction)**.**

$$\forall A \in \mathscr{F}. \quad \vdash A \implies \forall \rho.\, [\![A]\!]_\rho = 1$$

On en déduit immédiatement que :

Corollaire 2 (Cohérence)**.**

Dans le système logique, la formule $\perp$ n'est pas démontrable.

Démonstration :

Par définition on a bien $[\![\perp]\!] = [\![I(\texttt{false})]\!] = 0$ et par application contraposée du corollaire on conclut que cette formule n'est pas prouvable : $\nvdash \perp$. □

3.4. Bilan

Dans le modèle que nous venons de construire, l'appartenance d'un terme à l'interprétation d'un type lui garantit une forme forte de correction : l'assurance que, appelée dans un contexte adéquat (défini par le type), l'évaluation du programme terminera sur une valeur de la forme voulue.

Par ailleurs, on a montré la correction du moteur de preuve de PAF! relativement à ce modèle : toute formule que l'on peut dériver dans PAF! est vraie dans le modèle. On a donc montré que le typage fort entraîne la correction forte (la terminaison) et cela indépendamment de la vérification de type statique.

Dans la construction du modèle, nous avons aussi pu constater qu'il y avait plus de candidats que de types (incomplétude). Le modèle est suffisamment riche pour supporter de nombreuses extensions du système de types, telles que les types existentiels ou les types dépendants. Par exemple, le type (dépendant) des listes de longueur k serait naturellement interprété dans le modèle par l'ensemble

$$L_k = \bigcup_{n_1,\ldots,n_k \in [\![\mathtt{nat}]\!]^k \cap \mathbb{V}^k} \uparrow\{\,[n_1,\ldots,n_k]\,\}$$

des termes qui se réduisent sur un vecteur d'entiers de longueur k.

4. Le pendant sémantique du typage statique

Dans cette section nous donnons une sémantique au typage statique du système F à la Curry, étendu aux termes de notre langage, en définissant une nouvelle interprétation pour les types.

Le typage :

$$\frac{}{\Delta, x:\tau \vdash x:\tau}\,\text{AX} \qquad \frac{}{\Delta \vdash () : \mathtt{unit}}\,\text{UNIT} \qquad \frac{\Delta \vdash t:\tau' \qquad \Delta, x:\tau' \vdash u:\tau}{\Delta \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u:\tau}\,\text{LETIN}$$

$$\frac{\Delta \vdash t:\tau' \to \tau \qquad \Delta \vdash u:\tau'}{\Delta \vdash (t)u:\tau}\,\text{APP} \qquad \frac{\Delta, x:\tau \vdash t:\tau'}{\Delta \vdash \mathtt{fun}\ x \to t : \tau \to \tau'}\,\text{FUN} \qquad \frac{\Delta \vdash t:\tau \to \tau}{\Delta \vdash \mathtt{fix}\ t:\tau}\,\text{FIX}$$

$$\frac{\Delta \vdash t:\tau}{\Delta \vdash t:\forall\alpha.\tau}\,{\scriptstyle x \notin FV(\Delta)}\ \forall i \qquad \frac{\Delta \vdash t:\forall\alpha.\tau}{\Delta \vdash t:\tau[\tau'/\alpha]}\,\forall e$$

$$\frac{\Delta \vdash t:\tau \qquad \Delta \vdash u:\tau}{\Delta \vdash t = u : \mathtt{bool}}\,\text{EQUAL} \qquad \frac{\Delta \vdash b:\mathtt{bool} \qquad \Delta \vdash t:\tau \qquad \Delta \vdash u:\tau}{\Delta \vdash \mathtt{if}\ b\ \mathtt{then}\ t\ \mathtt{else}\ u:\tau}\,\text{IF}$$

$$\frac{}{\Delta \vdash \mathtt{true} : \mathtt{bool}}\,\text{TRUE} \qquad \frac{}{\Delta \vdash \mathtt{false} : \mathtt{bool}}\,\text{FALSE} \qquad \frac{}{\Delta \vdash 0 : \mathtt{nat}}\,\text{ZERO} \qquad \frac{}{\Delta \vdash [\,] : \forall\alpha.\ (\alpha\ \mathtt{list})}\,\text{EMPTY}$$

$$\frac{\Delta \vdash n:\mathtt{nat}}{\Delta \vdash \mathtt{succ}(n):\mathtt{nat}}\,\text{SUCC} \qquad \frac{\Delta \vdash n:\mathtt{nat} \qquad \Delta \vdash t:\tau \qquad \Delta \vdash u:\tau}{\Delta \vdash (\mathtt{match}_n\ n\ \mathtt{with}\ \ 0 \to t \mid \mathtt{succ}(x) \to u):\tau}\,\text{NAT}\mathbf{F}$$

$$\frac{\Delta \vdash t:\tau \qquad \Delta \vdash l:\tau\ \mathtt{list}}{\Delta \vdash (t :: l):\tau\ \mathtt{list}}\,\text{CONS} \qquad \frac{\Delta \vdash l:\tau\ \mathtt{list} \qquad \Delta \vdash t:\tau \qquad \Delta \vdash u:\tau}{\Delta \vdash (\mathtt{match}_l\ l\ \mathtt{with}\ \ [\,] \to t \mid x :: y \to u):\tau}\,\text{LIST}\mathbf{F}$$

où Δ désigne un contexte de typage qui associe des types clos aux variables de termes.

4.1. Le typage sémantique faible

La sémantique du typage statique se distingue par le fait que l'interprétation des types comprend des programmes divergents. Comme l'interprétation forte, l'*interprétation faible* est définie relativement à une valuation. Ici aussi, les valuations associent des termes clos aux variables de termes. En revanche, il est nécessaire de réadapter la définition des candidats :

Définition 10.

Un *candidat faible* est un ensemble de termes clos qui est lui-même clos par β-réduction et β-expansion. L'ensemble des candidats faibles est noté $\mathbb{C}^w$.

Remarque : contrairement aux candidats forts, les candidats faibles ne sont pas nécessairement l'expansion d'un ensemble de valeurs, *i.e.* un programme appartenant à un candidat faible ne termine

pas forcément sur une valeur. On voit donc que tout candidat (fort) est aussi un candidat faible. Comme précédemment, il y a plus de candidats faibles que de types qui peuvent être définis dans le langage.

Les valuations faibles (notée σ) sont définies sur les variables de termes, auxquelles elles associent des termes clos et sur les variables de type, auxquelles elles associent des candidats faibles. De la remarque précédente, on déduit qu'une valuation au sens fort est aussi une valuation au sens faible.

L'interprétation faible d'un terme t dans une valuation σ, notée $\{\!\{t\}\!\}_\sigma$, est définie de la même manière que précédemment par $t[\sigma]$ en considérant la restriction de σ aux variables de termes. Par contre, l'interprétation des types est modifiée pour ne plus tenir compte de la terminaison.

Définition 11 (Interprétation faible des types)**.**

$$\begin{array}{lcl}
\{\!\{\alpha\}\!\}_\sigma & = & \sigma(\alpha) \\
\{\!\{\texttt{bool}\}\!\}_\sigma & = & \{t \in \mathbb{T} \mid (\exists t'.t \rightsquigarrow t' \not\rightsquigarrow) \Rightarrow t' = \texttt{true} \vee t' = \texttt{false})\} \\
\{\!\{\texttt{nat}\}\!\}_\sigma & = & \bigcup_{n\in\mathbb{N}} \{t \in \mathbb{T} \mid (\exists t'.t \rightsquigarrow t' \not\rightsquigarrow) \Rightarrow t' = \texttt{succ}^n(0))\} \\
\{\!\{\tau\ \texttt{list}\}\!\}_\sigma & = & \bigcup_{n\in\mathbb{N}} \{t \in \mathbb{T} \ \mid\ (\exists t'.t \rightsquigarrow t' \not\rightsquigarrow) \Rightarrow \exists v_1, ..., v_n \in \{\!\{\tau\}\!\}_\sigma \cap \mathbb{V}\ t' = [v_1, ..., v_n])\} \\
\{\!\{\tau \to \tau'\}\!\}_\sigma & = & \{t \in \uparrow\mathbb{V} \ \mid\ \forall u \in \{\!\{\tau\}\!\}_\sigma \quad (t)u \in \{\!\{\tau'\}\!\}_\sigma\} \\
\{\!\{\forall\alpha.\tau\}\!\}_\sigma & = & \bigcap_{e\in\mathbb{C}^w} (\{\!\{\tau\}\!\}_{\sigma;\alpha\mapsto e})
\end{array}$$

Cette interprétation induit une forme de *correction faible*, qui ne garantit pas la terminaison mais garantit que, dans la mesure où l'on respecte les contextes d'application, l'évaluation ne produira pas d'erreur : si elle s'arrête, c'est sur une valeur (faiblement) correcte du type attendu, et non sur un terme « bloquant » ($t \rightsquigarrow t' \not\rightsquigarrow$ et $t' \notin \mathbb{V}$), ou sur une valeur d'un autre type. Dès lors, on constate que n'importe quel programme divergent est faiblement correct pour n'importe quel type : il appartient trivialement à l'interprétation faible de ce type. De même, les valeurs fonctionnelles dont l'application diverge, appartiennent à l'interprétation de n'importe quel type fonctionnel : ainsi `boucle` appartient à $\{\!\{\texttt{nat} \to \texttt{nat}\}\!\}$ [4].

Proposition 4.

Quel que soit le type τ et la valuation σ, l'interprétation de τ dans σ est un candidat faible : $\{\!\{\tau\}\!\}_\sigma \in \mathbb{C}^w$.

On peut maintenant montrer que le typage statique d'un terme entraîne sa correction faible :

Théorème 2 (Correction du typage statique)**.**

Soit $\tau, \tau_1, \ldots, \tau_n$ des types, t un terme dont les variables libres sont $x_1, \ldots, x_n$ et σ une valuation telle que pour tout i ($1 \leq i \leq n$), $\sigma(x_i) \in \{\!\{\tau_i\}\!\}_\sigma$.

Si le jugement de typage $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$ est dérivable, alors $\{\!\{t\}\!\}_\sigma \in \{\!\{\tau\}\!\}_\sigma$.

4.2. Observations

Un modèle ? On pourrait croire que cette interprétation peut être étendue en un modèle de PAF! Ce n'est cependant pas le cas. En effet :

Soit $f : \mathbb{T} \longrightarrow \{0, 1\}$ une fonction définie sur les termes clos et à valeurs dans $\{0, 1\}$ telle que :

$$f(t) = \begin{cases} 1 & \text{si } t \rightsquigarrow \texttt{true} \vee t \rightsquigarrow \texttt{false} \\ 0 & \text{sinon} \end{cases}$$

4. Comme dans la section précédente, lorsque le type interprété est clos on s'autorise à omettre la valuation.

On vérifie que f est une fonction propositionnelle β-compatible qui est vraie pour `true` ($f(\texttt{true}) = 1$) et `false`, mais fausse pour un booléen qui boucle ((`boucle` 0) par exemple) et on constate que la règle CASE-`bool` est invalidée.

Correction forte entraîne correction faible ? Nous sommes munis d'un modèle dans lequel les types sont interprétés de deux manières différentes permettant ainsi de caractériser deux notions distinctes de corrections des programmes. Ce modèle constitue un nouvel outil permettant de confronter ces deux notions de corrections et à travers elles, les techniques de typages sous-jacentes qu'elles reflètent.

Clairement, la correction faible d'un terme n'entraîne pas sa correction forte : `boucle` appartient à $\{\!\{\texttt{nat} \to \texttt{nat}\}\!\}$, mais n'appartient pas à $[\![\texttt{nat} \to \texttt{nat}]\!]$. Il pourrait en revanche sembler naturel que la correction forte d'un terme implique sa correction faible (pour tout τ clos, $[\![\tau]\!] \subset \{\!\{\tau\}\!\}$). La question n'est pourtant pas tranchée et dépend strictement du langage de programmation sous-jacent. En effet, si l'on considère un langage de programmation avec exceptions par exemple, alors l'implication est fausse :

```
let foo x = try (x 0) + 1 with
        | Not_found -> false
```

La fonction `foo` appartient à $[\![(\texttt{nat} \to \texttt{nat}) \to \texttt{nat}]\!]$, pour autant elle n'appartient pas à $\{\!\{(\texttt{nat} \to \texttt{nat}) \to \texttt{nat}\}\!\}$. Soit `bar1` une fonction fortement correcte dans $[\![\texttt{nat} \to \texttt{nat}]\!]$. Alors en évaluant `(foo bar1)` aucune exception ne sera levée lors du `try with` et le calcul renverra bien un entier naturel. En revanche, si l'on considère la fonction `bar2` suivante : `bar2 = fun y -> raise Not_found`, alors l'application `(foo bar2)` termine sur une valeur booléenne.

De même, en présence d'une égalité physique définie sur les fonctions, on peut écrire le programme suivant : `let F f = if (f == boucle) then false else (f 42)`. Pour toute fonction `f` dans $[\![\texttt{nat} \to \texttt{nat}]\!]$, la condition sera fausse (puisque `boucle` n'est pas dans cet ensemble), et `(F f)` renverra un entier naturel : $\texttt{F} \in [\![(\texttt{nat} \to \texttt{nat}) \to \texttt{nat}]\!]$. Par contre, `F` n'appartient pas à $\{\!\{(\texttt{nat} \to \texttt{nat}) \to \texttt{nat}\}\!\}$ puisqu'il existe au moins une fonction faiblement correcte `g` dans $\{\!\{\texttt{nat} \to \texttt{nat}\}\!\}$ telle que `(F g)` n'est pas dans $\{\!\{\texttt{nat}\}\!\}$: il s'agit de `boucle` elle-même. Néanmoins, pour le cas restreint du langage de programmation que nous avons présenté, nous conjecturons que c'est bien le cas.

Conjecture 1.

Quel que soit le type clos τ et la valuation ρ, on a $[\![\tau]\!]_\rho \subset \{\!\{\tau\}\!\}_\rho$

5. Bilan et Perspectives

Nous avons présenté une extension du système logique de PAF! (incluant les types du système F) où le typage statique a été retiré. Nous avons construit un modèle de réalisabilité très simple vis-à-vis duquel la correction de ce système a été montrée établissant ainsi sa cohérence. Le modèle a permis de donner une sémantique aux deux notions de typage, statique et fort, et constitue un cadre pertinent pour étudier leurs liens. De ce point de vue, l'inclusion des types forts dans les types faibles, pour le langage restreint que nous considérons (notamment sans exceptions et sans égalité physique), reste à démontrer (ou à invalider).

Extensions du système Plusieurs extensions du système (notamment au niveau de l'algèbre de types) peuvent être effectuées sans danger vis-à-vis de la cohérence. Le modèle garantit par exemple que l'extension aux types dépendants reste correcte. En particulier, les techniques de *type-based termination* pourraient être ajoutées au système. Cela diminuerait le nombre de preuves à charge de l'utilisateur, mais compliquerait significativement le système.

Outils d'automatisation Il est possible (et même souhaitable) d'intégrer des outils d'automatisation à PAF! Plusieurs tactiques offrent déjà la possibilité d'automatiser certaines démonstrations, en particulier les preuves de totalité de fonctions définies par récursion structurelle. Pour le moment, les fonctions récursives non structurelles se prouvent « à la main », mais rien n'empêche d'ajouter de nouvelles tactiques qui gèrent ces cas plus complexes.

Par ailleurs, il a été montré (Lemme 1) qu'une dérivation de typage dans le système F pouvait être transformée en une preuve de typage fort dans PAF! Il serait intéressant de réutiliser le typage statique, dont on avait jusque-là fait abstraction, et d'intégrer une nouvelle tactique qui se charge de faire cette transformation.

Programmes impératifs Nous travaillons actuellement sur une extension de PAF! qui intègre des constructions impératives : références, boucles et effets de bord. Il devient dès lors plus pertinent de prendre en considération les programmes dont l'évaluation ne s'arrête pas. À cet égard, la correction faible pourrait éventuellement se révéler adéquate, notamment dans la perspective d'adapter la preuve de cohérence que nous avons présentée dans cet article au nouveau système.

Références

[1] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4) :277–319, 2004. Special Issue : Fixed Points in Computer Science (FICS'03).

[2] Sylvain Baro. *Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML.* PhD thesis, PPS, Université Paris VII – Denis Diderot, Paris, France, 2003.

[3] S. Baro et P. Manoury. Un système x. raisonner formellement sur les programmes ml. *JFLA*, 2003.

[4] P. Taylor J-Y. Girard, Y. Lafont. *Proofs and types.* Cambridge University Press, 1989.

[5] J-L. Krivine. *Lambda-calcul, types et modèles.* Masson, 1990.

[6] C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[7] J. Wells. Typability and type checking in the second order lambda-calculus are equivalent and undecidable. *LICS*, 1994.

Remerciements Je tiens tout particulièrement à remercier les rapporteurs anonymes dont les critiques constructives m'ont permis d'améliorer ce travail. Je remercie également Pascal Manoury pour son aide précieuse et ses conseils éclairés.

Extraction certifiée dans Coq-en-Coq

Stéphane Glondu

Laboratoire Preuves, Programmes et Systèmes,
Université Paris Diderot - Paris 7,
Case 7014, 75205 Paris CEDEX 13, France
Stephane.Glondu@pps.jussieu.fr

Résumé

L'assistant de preuves Coq permet la génération de programmes corrects par construction. Cette fonctionnalité, appelée extraction, est exploitée notamment pour produire des bibliothèques de fonctions certifiées. Nous présentons dans cet article une formalisation de l'extraction en Coq, ainsi que certaines de ses propriétés qui ont été prouvées formellement. Ces travaux s'inscrivent dans le cadre de la formalisation de Coq en Coq de B. Barras.

1. Introduction

Programmation fonctionnelle et λ-calcul

Les langages fonctionnels — tels qu'OCaml, Haskell ou Scheme — sont des langages dérivés du λ-calcul. Dès 1958, H. Curry remarque dans [4] une analogie entre le λ-calcul simplement typé et la logique propositionnelle intuitionniste, établissant ainsi un isomorphisme entre types et formules, et entre programmes et preuves. En 1969, W. A. Howard étend cet isomorphisme à la logique du premier ordre en proposant un λ-calcul avec types dépendants [8]. Cet isomorphisme est depuis couramment connu sous le nom d'*isomorphisme de Curry-Howard*.

La formalisation des systèmes logiques a donné naissance à des *assistants de preuves*, logiciels vérifiant la validité de preuves. Avec l'isomorphisme de Curry-Howard en tête, il est tentant de déterminer le contenu calculatoire des preuves, d'*extraire* des programmes à partir des preuves. Les programmes extraits sont alors naturellement exprimés dans des langages fonctionnels. Cette idée a été mise en pratique dans de nombreux systèmes tels que Nuprl [10], Minlog [6] ou Isabelle [2, 3]. Nous nous intéressons ici plus particulièrement à Coq.

Les assistants de preuves au service de la certification

Les assistants de preuves permettent de prouver formellement des théorèmes mathématiques « usuels », mais aussi des propriétés sur des programmes et des systèmes informatiques en tous genres. Ils sont parfois utilisés pour prouver des propriétés sur des programmes impératifs, mais en général seuls certains aspects des langages impératifs sont considérés, et il n'est pas rare de devoir supposer un nombre important d'hypothèses concernant le langage avant de pouvoir prouver des énoncés.

En revanche, les assistants de preuves tels que Coq ou Isabelle permettent de définir des fonctions dans un style fonctionnel et de s'en servir directement dans l'énoncé de théorèmes. Ces fonctions peuvent ensuite se traduire — disons plutôt *s'extraire* — directement vers un langage fonctionnel.

En outre, considérons une preuve (constructive) de l'énoncé suivant :

$$\forall a, b \in \mathbb{N}, \quad b \neq 0 \implies \exists q, r \in \mathbb{N}, \quad a = bq + r \quad \text{et} \quad 0 \leqslant r < b.$$

Derrière cette preuve se cache un programme de division euclidienne, et c'est le rôle de l'extraction de l'exhiber. Le programme (fonctionnel) ainsi obtenu, prenant un a et un b, et retournant un q et un r, vérifiera alors naturellement l'énoncé ci-dessus, pour peu que l'extraction soit correcte.

L'extraction en Coq

Coq est bâti directement sur la correspondance de Curry-Howard : toutes les preuves sont en interne représentées comme des termes du Calcul des Constructions Inductives, un dérivé du λ-calcul. On peut donc dire que dans ce formalisme, toutes les preuves *sont* des programmes. Cependant, dans ces programmes, on peut distinguer des parties purement logiques et des parties vraiment calculatoires, distinction qui peut déjà être sentie dans certains énoncés : ainsi, dans une proposition existentielle telle que « $\exists x, P(x)$ », on est souvent intéressé par la manière dont le x est construit ; concernant l'énoncé $P(x)$, on a juste besoin de savoir qu'il est vrai, mais la plupart du temps on ne veut pas savoir pourquoi.

L'extraction en Coq a pour rôle de séparer ces différentes composantes, d'effacer les parties logiques afin de ne laisser que les parties calculatoires des preuves. En Coq, cela est réalisé en mettant les propositions purement logiques dans une classe de types particulière, `Prop`. Ainsi, dans l'énoncé « $\exists x, P(x)$ », « $P(x)$ » sera typiquement de type `Prop`, alors que x pourra avoir un vrai type de données. Après effacement de ces parties logiques, il est souhaitable de garder une relation entre le programme extrait et la proposition de départ, afin de pouvoir bénéficier de ce qui a été prouvé. À cette fin, on peut utiliser la notion de réalisabilité, introduite par S. C. Kleene en 1945 dans [9].

L'extraction en Coq a commencé avec la thèse de C. Paulin [13, 14, 15], qui a défini une relation de réalisabilité adaptée au Calcul des Constructions, à la base du Coq de l'époque, et une extraction associée. Cette extraction théorique a été prouvée correcte dans [14] et a été implantée dans Coq. Cependant, l'extraction de C. Paulin souffrait de quelques restrictions. Ces restrictions, ainsi que l'évolution de Coq vers le Calcul des Constructions *Inductives*, ont mené P. Letouzey à proposer une nouvelle extraction pour Coq dans sa thèse [12]. Cette extraction a été implantée dans Coq, et c'est elle qui est actuellement utilisée.

Une extraction certifiée ?

L'extraction est actuellement un programme écrit en OCaml, pour lequel aucune preuve formelle n'a été réalisée. Cette extraction est issue des travaux de P. Letouzey, qui a prouvé « sur le papier » des résultats théoriques sur ce processus. Cependant, lorsqu'on travaille avec des assistants de preuves, la tentation est grande de prouver de tels résultats en Coq. L'objectif des travaux présentés ici était de voir dans quelle mesure ces résultats pouvaient être transposés en Coq.

Cette étape est importante dans la validation de toute une chaîne de compilation entamée d'une part par B. Barras avec le développement en Coq d'un noyau comparable à celui de Coq [1], et d'autre part par Z. Dargaye avec le développement d'un compilateur de ML certifié [5].

Contribution

Nous présentons ici une extraction formalisée en Coq. B. Barras a, dans sa thèse [1], formalisé le Calcul des Constructions Inductives — la théorie logique à la base de Coq — en Coq. Ce développement a conduit par extraction à un noyau certifié utilisé par un système de preuves minimal. Comme le souligne B. Barras à la fin de sa thèse, une formalisation de l'extraction est nécessaire afin de pouvoir réaliser le *bootstrap* d'un système de preuves assimilable à Coq. Nous avons repris les travaux de P. Letouzey dans le formalisme de B. Barras, ajoutant ainsi une extraction au système de preuves certifié déjà réalisé. Un des principaux théorèmes de P. Letouzey relatif à l'extraction a été prouvé. Le

développement est disponible sur le site web de l'auteur. La partie concernant l'extraction proprement dite fait environ 1 200 lignes de Coq et a demandé quatre mois de développement.

Nous présentons brièvement, dans la section 2, la formalisation de Coq de B. Barras, puis l'extraction proprement dite dans la section section 3. Afin de rendre la lecture plus agréable, nous avons choisi d'écrire beaucoup d'énoncés Coq dans un style mathématique. Pour aider le lecteur à mettre en correspondance les énoncés de ce rapport avec les développements Coq, nous donnons les dénominations Coq des définitions et théorèmes en style `machine à écrire`.

2. Coq-en-Coq

2.1. Les Systèmes de Types Purs

B. Barras conçoit un système de preuves où celles-ci sont représentées par des termes d'un langage typé dérivé du λ-calcul, le *Calcul des Constructions Inductives* (CCI), mettant ainsi en pratique l'isomorphisme de Curry-Howard. C'est, à quelques détails près, le même CCI qui est mis en œuvre dans Coq. Cependant, le CCI et la certification d'un noyau comparable à celui de Coq n'est que l'aboutissement d'un développement commençant avec le λ-calcul et passant par les Systèmes de Types Purs (PTS) avec divers enrichissements. Nous supposerons le lecteur familier avec le λ-calcul typé ainsi qu'avec Coq, et nous présenterons ici rapidement les PTS avec sous-typage et opérateurs (PTSO). Nous invitons le lecteur curieux à consulter les chapitres 5 et 6 de [1] pour plus de détails.

2.1.1. Langage

Les termes sont paramétrés par trois ensembles sur lesquels l'égalité est décidable :
- un ensemble de *sortes* `sort`. Une sorte est une constante particulière, qui est le type d'une certaine classe de types ;
- un ensemble non vide de *noms* `name` (nous noterons `_` un élément particulier de cet ensemble). Ces noms seront notamment utilisés pour le confort de l'utilisateur final (saisie, affichage) pour nommer les variables liées dans un terme, bien que des indices de de Bruijn soient utilisés en interne. Cependant, les noms seront significatifs par la suite lorsque nous introduirons les opérateurs pour exprimer les constantes globales et les inductifs ;
- un ensemble d'*opérateurs* `oper`. Pour éviter d'avoir à trop étendre son langage pour pouvoir tenir compte de toutes les constructions du CCI alors que certaines propriétés ne sont pas spécifiques au CCI, B. Barras utilise cette notion d'opérateur, qui est en quelque sorte une constante équipée d'un schéma de type et éventuellement de règles de réduction.

Définition 2.1 (type `term`)**.** Le type des *termes* est défini par la grammaire suivante :

$$\begin{array}{rcl} T_1, T_2 : \texttt{term} & := & s \mid \natural n \mid c \mid c(T_1) \\ & & \mid \Pi x : T_1.T_2 \mid \lambda x : T_1.T_2 \\ & & \mid T_1 * T_2 \mid (T_1, T_2) \end{array}$$

où $s \in \texttt{sort}$, $x \in \texttt{name}$, $c \in \texttt{oper}$, et $n \in \mathbb{N}$.

Remarquons qu'il n'y a pas de construction générique pour l'application, mais qu'il y en a une pour les opérateurs d'arité zéro ou un, les arités supérieures étant simulées à l'aide des paires (T_1, T_2). L'application usuelle sera ainsi un opérateur prenant en argument un couple (f, x) d'une fonction et d'un argument. Cette présentation permet ainsi de traiter le filtrage d'une façon similaire à l'application. Les sommes $T_1 * T_2$ représentent les types des couples (T_1, T_2), de la même manière que les produits $\Pi x : T_1.T_2$ représentent les types des abstractions $\lambda x : T_1.T_2$. Les constructions $\lambda x : T_1.T_2$ et $\Pi x : T_1.T_2$ sont les seules à lier des variables, et seront utilisées par des opérateurs

pour fournir des constructions plus évoluées introduisant des liaisons. Nous rappelons ici que x est purement décoratif, et $\natural n$ représente la variable d'indice de de Bruijn n. De manière générale, nous désignerons par s une sorte, par x un nom et par c un opérateur.

Comme il est de coutume lorsqu'on parle de λ-calcul, on définit l'opération de *substitution* sur les termes :

$$t\,\{\natural 0 \leftarrow u\}$$

désigne le terme t dans lequel toutes les occurrences de $\natural 0$ ont été remplacées par u. Derrière cette description informelle se cache une définition très technique, que nous ne détaillerons pas ici.

Cette notion de substitution va de pair avec la notion d'environnement.

Définition 2.2 (types `env`, `decl`)**.** Un *environnement* est une liste de *déclarations*, et est défini par la grammaire suivante :

$$\Gamma : \texttt{env} \quad := \quad [] \mid \Gamma[x : T] \mid \Gamma[x \doteq t : T]$$

Lorsque δ est une déclaration, nous désignerons par x_δ son nom, par T_δ son type et par t_δ son corps (si elle en a un).

Intuitivement, dans un environnement Γ, la variable $\natural 0$ fait référence à la dernière déclaration de Γ.

En outre, nous noterons $\uparrow_k^n t$ le terme t où toutes les variables libres sous k lieurs de t sont incrémentées de n. Cette opération est appelée *relocation*.

2.1.2. Signature et typage

Un *jugement de typage* aura la forme $\Gamma \vdash t : T$, signifiant que t a le type T dans l'environnement Γ. t et T sont deux termes du CCI : contrairement au λ-calcul simplement typé, les termes et les types vivent dans le même espace. Dans le même style que les termes, la définition des jugements de typage sera paramétrique, et sera instanciée plus tard au CCI.

Définition 2.3 (type `PTSO_spec`)**.** La spécification d'un PTS avec sous-typage et opérateurs (ou *PTSO*) est une structure regroupant six paramètres :

$$\begin{array}{ll} \langle & \texttt{axiom}: \quad \mathcal{P}(\texttt{sort} \times \texttt{sort}); \\ & \texttt{rule}: \quad \mathcal{P}(\texttt{sort} \times \texttt{sort} \times \texttt{sort}); \\ & \texttt{pair}: \quad \mathcal{P}(\texttt{sort} \times \texttt{sort} \times \texttt{sort}); \\ & \leqslant: \quad \mathcal{P}(\texttt{env} \times \texttt{term} \times \texttt{term}); \\ & \Sigma_0: \quad \mathcal{P}(\texttt{oper} \times \texttt{term}); \\ & \Sigma_1: \quad \mathcal{P}(\texttt{oper} \times \texttt{env} \times \texttt{term} \times \texttt{term} \times \texttt{term}) \\ \rangle & \end{array}$$

Ici, $\mathcal{P}(\texttt{X})$ désigne l'ensemble des parties de `X`, ce que l'on peut écrire `X -> Prop` en Coq.

Les rôles des différentes composantes devraient devenir clairs avec la prochaine définition. En réalité, la structure `PTSO_spec` de [1] est plus riche et comporte également des preuves de lemmes de compatibilité entre le sous-typage $\leqslant$, la signature Σ_1 et les opérations de substitution et relocation, lemmes que nous ne détaillerons pas ici.

Définition 2.4 (prédicats `wf`, `typ`)**.** Les règles de typage des contextes et des termes des PTSO sont les suivantes :

$$\frac{}{[] \vdash}\,\texttt{Wf_nil} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash T : s}{\Gamma[x : T] \vdash}\,\texttt{Wf_cons_var} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash t : T \quad \Gamma \vdash T : s}{\Gamma[x \doteq t : T] \vdash}\,\texttt{Wf_cons_def}$$

$$\frac{\Gamma \vdash \quad (s_1, s_2) \in \texttt{axiom}}{\Gamma \vdash s_1 : s_2}\,\texttt{Typ_srt} \qquad \frac{\Gamma \vdash \quad \Gamma(n) = \delta}{\Gamma \vdash \natural n : \uparrow_0^{n+1} T_\delta}\,\texttt{Typ_rel} \qquad \frac{\Gamma \vdash \quad [_ : T] \vdash \quad (c, T) \in \Sigma_0}{\Gamma \vdash c : T}\,\texttt{Typ_cst0}$$

$$\frac{\Gamma[_ : T] \vdash \quad \Gamma[_ : U] \vdash \quad \Gamma \vdash t : T \quad (c, \Gamma, t, T, U) \in \Sigma_1}{\Gamma \vdash c(t) : U}\ \texttt{Typ_cst1}$$

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma[x : T] \vdash U : s_2 \quad (s_1, s_2, s_3) \in \texttt{rule}}{\Gamma \vdash \Pi x : T.U : s_3}\ \texttt{Typ_prd}$$

$$\frac{\Gamma[x : T] \vdash M : U \quad \Gamma \vdash \Pi x : T.U : s}{\Gamma \vdash \lambda x : T.M : \Pi x : T.U}\ \texttt{Typ_lam}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash B : s_2 \quad (s_1, s_2, s_3) \in \texttt{pair}}{\Gamma \vdash A * B : s_3}\ \texttt{Typ_sum} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B \quad \Gamma \vdash A * B : s}{\Gamma \vdash (M, N) : A * B}\ \texttt{Typ_pair}$$

$$\frac{\Gamma \vdash M : U \quad \Gamma \vdash V : s \quad \Gamma \vdash U \leqslant V}{\Gamma \vdash M : V}\ \texttt{Typ_conv} \qquad \frac{\Gamma \vdash M : U \quad \Gamma \vdash U \leqslant s}{\Gamma \vdash M : s}\ \texttt{Typ_conv_srt}$$

2.1.3. Sous-typage et réduction

Le sous-typage, utilisé par les règles `Typ_conv` et `Typ_conv_srt`, regroupe dans le cas du CCI les règles de cumulativité et de conversion. C'est donc là qu'interviennent les règles de réduction. En fait, la relation $\leqslant$ donnée par la structure `PTSO_spec` fait appel à des opérations de clôture qui restent génériques. Dans le cadre de l'extraction, on ne s'intéressera guère à l'aspect typage du sous-typage, mais plutôt à l'aspect réduction. Par conséquent, nous ne détaillerons ici que les composantes calculatoires du sous-typage.

Remarque 2.5. Pratiquement toutes les notions présentées dans cet article dépendent d'un environnement. Cependant, afin d'alléger les notations, nous omettrons souvent les environnements. Parfois, ces environnements omis seront liés entre eux par des opérations complexes (insertions, relocations, substitutions). En cas de doute, le lecteur pourra se référer au développement Coq.

Définition 2.6 (opération `ctxt`)**.** Soit $\rightarrow$ une règle de réduction. Les règles suivantes définissent la *clôture par contexte* $\rightarrow_{\mathcal{X}}$ de $\rightarrow$:

$$\frac{t \rightarrow t'}{t \rightarrow_{\mathcal{X}} t'}\ \texttt{(W)Ctx_rule} \qquad \frac{t \rightarrow t'}{c(t) \rightarrow_{\mathcal{X}} c(t')}\ \texttt{(W)Ctx_cst}$$

$$\frac{T \rightarrow T'}{\lambda x : T.M \rightarrow_{\mathcal{X}} \lambda x : T'.M}\ \texttt{(W)Ctx_lam_l} \qquad \frac{M \rightarrow M'}{\lambda x : T.M \rightarrow_{\mathcal{X}} \lambda x : T.M'}\ \texttt{Ctx_lam_r}$$

$$\frac{T \rightarrow T'}{\Pi x : T.U \rightarrow_{\mathcal{X}} \Pi x : T'.U}\ \texttt{(W)Ctx_prd_l} \qquad \frac{U \rightarrow U'}{\Pi x : T.U \rightarrow_{\mathcal{X}} \Pi x : T.U'}\ \texttt{Ctx_prd_r}$$

$$\frac{A \rightarrow A'}{A * B \rightarrow_{\mathcal{X}} A' * B}\ \texttt{(W)Ctx_sum_l} \qquad \frac{B \rightarrow B'}{A * B \rightarrow_{\mathcal{X}} A * B'}\ \texttt{(W)Ctx_sum_r}$$

$$\frac{M \rightarrow M'}{(M, N) \rightarrow_{\mathcal{X}} (M', N)}\ \texttt{(W)Ctx_pair_l} \qquad \frac{N \rightarrow N'}{(M, N) \rightarrow_{\mathcal{X}} (M, N')}\ \texttt{(W)Ctx_pair_r}$$

Si dans un système, la réduction est close par contexte, elle sera qualifiée de *forte*.

Les `(W)` apparaissant dans la définition ci-dessus seront expliqués plus tard, lors de la définition 3.2.

2.2. Le Calcul des Constructions Inductives

Le CCI tel que présenté dans [1] est une instance de PTSO.

2.2.1. Sortes

Nous disposons d'une hiérarchie de sortes comprenant deux sortes imprédicatives `Prop` et `Set` [1], et une hiérarchie d'univers prédicatifs $\texttt{Type}_i$ ($i \in \mathbb{N}$). Nous ne détaillerons pas ici les règles de typage des sortes `axiom`, ni celles de formation des produits `rule`.

2.2.2. Opérateurs

C'est là que l'on retrouve toutes les constructions du CCI.

Définition 2.7 (type `cci_op`, instanciation de `oper`)**.** L'ensemble des opérateurs du CCI est le suivant :

$$\begin{array}{rcl} c : \texttt{cci_op} & := & \pi_1 \mid \pi_2 \mid @ \mid \texttt{Const}\,\{C\} \\ & & \mid \texttt{Ind}\,\{I, n\} \mid \texttt{Constr}\,\{C\} \mid \texttt{Case}\,\{\vec{p}\} \mid \texttt{Fix} \\ & & \mid \square \mid \varepsilon \mid ::\mid \mathcal{M} \mid \mathcal{L} \\ & & \mid \texttt{Record}\,\{\vec{x}\} \mid \texttt{Struct}\,\{\vec{x}\} \mid \texttt{Field}\,\{x\} \end{array}$$

où $n \in \mathbb{N}$, $I, C, x \in \texttt{name}$ et $\vec{p}, \vec{x} \in \texttt{name}^*$.

Nous avons dans l'ordre : les projections, l'application, la constante globale, le type inductif, le constructeur, le filtrage, le point fixe, cinq opérateurs liés aux *marques*, les enregistrements, l'opérateur correspondant à leur type, et l'accès à un certain champ d'un enregistrement. Les deux dernières lignes ne correspondent pas à des constructions de Coq [2].

Dans la définition précédente, les noms I, C et $\vec{p}$ font référence à un *environnement global* Δ, indexé par des noms. Cet environnement peut contenir des définitions, des axiomes, mais contient aussi la définition des inductifs. En réalité, B. Barras définit le CCI comme une famille de PTSO (`cci_pts`), indexée par Δ. Dans tous nos développements, nous supposerons Δ fixé, et donc nous travaillerons bien dans un PTSO.

Plutôt que de donner la définition formelle de Σ_0 et de Σ_1 (respectivement `mem_sign0` et `mem_sign` en Coq), nous allons dans la suite de cette section décrire informellement la signification des opérateurs.

(Non-)Curryfication B. Barras présente son langage d'une façon fortement non curryfiée : là où on s'attendrait à rencontrer plusieurs termes, il n'en met qu'un seul et exploite la construction de paire (_, _) des PTSO. Cela permet de simplifier considérablement certains énoncés Coq.

Marques Les *marques* jouent un rôle important dans le typage des points fixes que nous ne détaillerons pas ici. Il servent également à marquer une absence ou un délimiteur. $\square$ peut ainsi jouer le même rôle que `()` en OCaml, $\mathcal{M}$ est son type (tout comme `unit`), $\mathcal{L}$ est le type des listes de marques, et ε et :: ses constructeurs. Les quatre premiers de ces cinq opérateurs sont d'arité zéro (ce sont les seuls), le dernier étant unaire (prenant une paire en argument).

1. Cette hiérarchie de sortes correspond à celle de Coq 6.2. L'imprédicativité de `Set` étant incompatible avec certains axiomes raisonnables [7], elle a été retirée à partir de la version 8.0 : `Set` n'est plus qu'une sorte prédicative au même titre que les $\texttt{Type}_i$. Ce détail a une importance marginale dans le cadre de l'extraction.
2. En Coq, les enregistrements ne sont que du sucre syntaxique pour des inductifs à un constructeur.

Notations Écrire les termes formellement en utilisant les opérateurs et les paires peut être très pénible, aussi ne nous hasarderons-nous pas à utiliser systématiquement cette syntaxe [3]. Nous nous permettrons ainsi d'omettre des parenthèses et nous noterons :

- $(t_1, t_2, \ldots, t_n)$, ou $\vec{t}$ (dans un contexte de terme), le terme

$$(t_1, (t_2, \ldots (t_n, \varepsilon) \ldots));$$

- $f\,x$, le terme $@((f, x))$;
- $\texttt{Case}\,\{\vec{p}\}\,(t, P, \vec{f})$, le filtrage de l'objet t en utilisant le prédicat d'élimination P, et les branches sous forme fonctionnelle $\vec{f}$, $\vec{p}$ représentant la liste les noms des constructeurs du type (inductif) de t ;
- $\texttt{Fix}(f : T)$, où f est de la forme

$$\lambda F : T.\lambda p : U.\lambda x : V.M,$$

  la fonction récursive d'argument x de type inductif V (dépendant éventuellement de p de type U), de corps M (les appels récursifs se faisant avec la variable F), le type du point fixe résultant étant T ;
- $\langle \vec{x} \doteq \vec{t} : \vec{T} \rangle$, plus formellement $\texttt{Struct}\,\{\vec{x}\}\,(\vec{T}, \vec{t})$, un enregistrement avec les champs $\vec{x}$ de types $\vec{T}$ et de valeurs $\vec{t}$;
- $\langle \vec{x} : \vec{T} \rangle$, plus formellement $\texttt{Record}\,\{\vec{x}\}\,(\vec{T})$, le type correspondant ;
- $t_{\langle x \rangle}$, plus formellement $\texttt{Field}\,\{x\}\,(t)$, la projection sur x de l'enregistrement t ;

Points fixes Dans le CCI, les définitions de fonctions récursives (appelées points fixes) doivent explicitement mentionner un argument *structurellement* décroissant, afin de garantir leur bonne fondation. La vérification est faite syntaxiquement dans Coq. Dans le développement de B. Barras, le critère est sémantique, et intégré au typage (utilisant les marques), mais nous allons ici nous contenter de l'intuition que l'argument du point fixe doit décroître avec chaque appel récursif. Il est à noter qu'ici, l'argument d'un point fixe ne peut dépendre que d'un seul terme que nous appellerons *paramètre*. L'argument n'est pas non plus obligé de dépendre du paramètre. Cela reviendrait dans Coq à rassembler systématiquement en un seul tous les arguments qui précèdent celui qui décroît structurellement.

Constantes globales L'opérateur $\texttt{Const}\,\{x\}$ désigne une constante dont la définition est dans l'environnement global Δ sous le nom x. Il s'agit d'un opérateur unaire pouvant prendre en argument un *paramètre*, qui peut être référencé par $\natural 0$ dans le corps de x. Il ne faut pas confondre $\texttt{Const}\,\{x\}$ avec une fonction, mais plutôt le voir comme une constante paramétrée. L'intuition derrière ce choix est que le paramètre est généralement inférable et — la plupart du temps — ne devrait pas être explicité.

Inductifs L'opérateur $\texttt{Ind}\,\{I, n\}$ est un opérateur unaire, dont l'argument $(p, (a, l))$ contient le *paramètre* (au même sens que Coq) p et l'argument a de l'inductif. n (un entier) et l (une liste de marques) sont utilisés pour le typage des points fixes. La définition de l'inductif [4] est identifiée par I dans l'environnement global Δ. C'est dans Δ qu'est stockée la liste des constructeurs de I, ainsi que leurs types.

Constructeurs L'opérateur $\texttt{Constr}\,\{C\}$ est un opérateur unaire dont l'argument (p, a) contient le paramètre p de l'inductif associé et l'argument a du constructeur.

3. D'ailleurs, le système de notations de Coq est beaucoup sollicité dans le développement de Barras !
4. Les inductifs mutuels et les points fixes mutuels sont traités dans le développement Coq, mais nous écarterons silencieusement ces possibilités ici.

Remarque 2.8 (Distinction `Set`/`Prop`)**.** La sorte `Prop` est utilisée pour typer les propositions logiques, et la sorte `Set`, pour typer les types de données d'objets calculatoires. Intuitivement, l'utilité des propositions est de savoir qu'elles sont prouvées ; un objet calculatoire ne doit pas dépendre de la façon dont une proposition a été prouvée. Ce fait est d'ailleurs exploité dans l'extraction : toutes les preuves de propositions logiques seront effacées. Cela impose certaines contraintes sur les filtrages, que nous ne détaillerons pas ici.

2.2.3. Réduction

Pour avoir une instanciation complète de PTSO, nous devons aussi donner la règle de sous-typage $\leqslant$. Cette règle en soit joue un rôle mineur dans l'extraction, donc nous ne la présenterons pas ici. Cependant, elle contient la relation de réduction $\rightarrow_{\mathcal{C}}$ qui nous servira à établir la correction de l'extraction.

Définition 2.9 (prédicats `redn_term`, $\rightarrow_{\mathcal{C}}$)**.** La règle `redn_term` est l'union des règles suivantes :

$$\frac{\Gamma(n) = [x \doteq t : T]}{\Gamma \vdash \natural n \rightarrow_{\delta} \uparrow^{n+1} t}\ \texttt{delta} \qquad \frac{\Delta(x) = [x \doteq t : T]}{\texttt{Const}\,\{x\}\,(M) \rightarrow_{\Delta} t\,\{\natural 0 \leftarrow M\}}\ \texttt{delta_glob}$$

$$\frac{}{\pi_1(M,N) \rightarrow_{\pi_1} M}\ \texttt{proj1} \qquad \frac{}{\pi_2(M,N) \rightarrow_{\pi_2} N}\ \texttt{proj2} \qquad \frac{y = x_i}{\langle \vec{x} \doteq \vec{t} : \vec{T} \rangle_{\langle y \rangle} \rightarrow_{\pi_r} t_i}\ \texttt{projr}$$

$$\frac{}{(\lambda x : T.M)N \rightarrow_{\beta} M\,\{\natural 0 \leftarrow N\}}\ \texttt{beta}$$

$$\frac{t = \texttt{Constr}\,\{C\}\,(M)}{\texttt{Fix}(f : T)\ p\ t \rightarrow_{\iota_{\texttt{fix}}} f\ (\texttt{Fix}(f : T))\ p\ t}\ \texttt{iota_fix} \qquad \frac{t = \texttt{Constr}\,\{C\}\,(M) \qquad p_i = C}{\texttt{Case}\,\{\vec{p}\}\,(t, P, \vec{f}) \rightarrow_{\iota_{\texttt{case}}} f_i\,M}\ \texttt{iota_case}$$

On notera $\rightarrow_{\mathcal{C}}$ la clôture par contexte de `redn_term`.

Petit point sur les notations « $\Gamma \vdash t \rightarrow_{\mathcal{C}} u$ » désigne ce que l'on note en Coq « `ctxt redn_term` $\Gamma\ t\ u$ ». Il s'agit de *la* règle correspondant à une étape de réduction du CCI.

3. Extraction vers CCI$_{\square}$

3.1. Prédicat d'extraction partielle

L'extraction consiste à transformer un terme CCI en un terme ML (par exemple) en effaçant toutes les annotations de type, ainsi que les preuves de propositions logiques. Cette opération ne devrait pas changer la sémantique du terme s'il calcule effectivement quelque chose.

On pourrait être tenté de définir directement une fonction d'extraction d'un terme CCI vers un terme ML. Cependant, comme l'explique Letouzey dans [12], une telle fonction ne présente pas de bonnes propriétés et cette approche n'est pas suffisamment générale si l'on veut prouver la correspondance entre les réductions du terme original et celles du terme extrait. Par conséquent, nous utiliserons plutôt une relation pour modéliser une extraction « partielle ».

Définition 3.1 (prédicat `Pe`)**.** La relation d'extraction $\Gamma \vdash t \rightarrow_{\mathcal{E}} t'$ est définie par les règles suivantes :

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : \texttt{Prop}}{\Gamma \vdash t \rightarrow_{\mathcal{E}} \square}\ \texttt{Pe_Prop} \qquad \frac{t \in \texttt{sort}}{t \rightarrow_{\mathcal{E}} \square}\ \texttt{Pe_Srt} \qquad \frac{t \in \texttt{oper}}{t \rightarrow_{\mathcal{E}} \square}\ \texttt{Pe_Cst0}$$

$$\frac{}{t_1 * t_2 \to_{\mathcal{E}} \square}\ \texttt{Pe_Pair} \qquad \frac{}{\Pi x : T.t \to_{\mathcal{E}} \square}\ \texttt{Pe_Prd} \qquad \frac{}{\natural n \to_{\mathcal{E}} \natural n}\ \texttt{Pe_Rel}$$

$$\frac{c \in \{\texttt{Ind}\,\{I, n\}\,, \texttt{Record}\,\{\vec{x}\}\}}{c(t) \to_{\mathcal{E}} \square}\ \texttt{Pe_\{MutInd,Record\}} \qquad \frac{t \to_{\mathcal{E}} t' \quad i \in \{1, 2\}}{\pi_i(t) \to_{\mathcal{E}} \pi_i(t')}\ \texttt{Pe_Proj}i$$

$$\frac{t_1 \to_{\mathcal{E}} t'_1 \qquad t_2 \to_{\mathcal{E}} t'_2}{t_1\, t_2 \to_{\mathcal{E}} t'_1\, t'_2}\ \texttt{Pe_App} \qquad \frac{t_1 \to_{\mathcal{E}} t'_1 \qquad t_2 \to_{\mathcal{E}} t'_2}{(t_1, t_2) \to_{\mathcal{E}} (t'_1, t'_2)}\ \texttt{Pe_Pair}$$

$$\frac{t \to_{\mathcal{E}} t'}{\lambda x : T.t \to_{\mathcal{E}} \lambda x : \square.t'}\ \texttt{Pe_Lam} \qquad \frac{f \to_{\mathcal{E}} f'}{\texttt{Fix}(f : T) \to_{\mathcal{E}} \texttt{Fix}(f' : \square)}\ \texttt{Pe_Fix}$$

$$\frac{t \to_{\mathcal{E}} t' \qquad f \to_{\mathcal{E}} f'}{\texttt{Case}\,\{\vec{p}\}\,(t, P, f) \to_{\mathcal{E}} \texttt{Case}\,\{\vec{p}\}\,(t', \square, f')}\ \texttt{Pe_Case}$$

$$\frac{t \to_{\mathcal{E}} t' \qquad c \in \{\texttt{Const}\,\{x\}\,, \texttt{Constr}\,\{C\}\}}{c(t) \to_{\mathcal{E}} c(t')}\ \texttt{Pe_\{Const,Constr\}}$$

$$\frac{\vec{t} \to_{\mathcal{E}} \vec{t'}}{\langle \vec{x} \doteq \vec{t} : \vec{T} \rangle \to_{\mathcal{E}} \langle \vec{x} \doteq \vec{t'} : \square \rangle}\ \texttt{Pe_Struct} \qquad \frac{t \to_{\mathcal{E}} t'}{t_{\langle x \rangle} \to_{\mathcal{E}} t'_{\langle x \rangle}}\ \texttt{Pe_Projr}$$

On dira alors que t s'*extrait* (partiellement) vers t' dans le contexte Γ.

Dans cette définition, on peut distinguer trois types de règles :
- `Pe_Prop` élimine les parties logiques ;
- les autres règles produisant $\square$ éliminent ce qui est lié au typage, comme les opérateurs sans argument (`Pe_Cst0`) ou les types eux-mêmes ;
- les règles restantes ne font que propager l'extraction aux sous-termes, en prenant soin d'effacer les types.

La relation $\to_{\mathcal{E}}$ n'est pas fonctionnelle : bien que la plupart des règles soient dirigées par la syntaxe, la règle `Pe_Prop` peut intervenir n'importe où. Un chevauchement est ainsi possible entre les règles.

Comme mentionné à la remarque 2.5, nous omettrons souvent le « $\Gamma \vdash$ ». Notons que si le terme t' reste syntaxiquement un terme de CCI, il appartient à un sous-langage plus proche du λ-calcul non typé que du CCI, langage que nous noterons $\text{CCI}_\square$. C'est ce qui jouera le rôle de notre ML. Nous avons choisi de conserver la même syntaxe afin d'alléger le développement. Les $\square$ introduits représentent les parties éliminées par l'extraction. Il est important de remarquer que CCI et $\text{CCI}_\square$ ne partagent que de la syntaxe : en général, t' n'est pas un terme correctement typé du CCI. De plus, nous allons munir $\text{CCI}_\square$ de règles de réductions différentes de celles du CCI, afin de mieux correspondre à ML.

3.2. Réduction dans $\text{CCI}_\square$

Nous décrivons ici les réductions du $\text{CCI}_\square$. Ces réductions correspondent à un langage de programmation fonctionnelle tel que ML. Contrairement au CCI, les sous-termes d'une abstraction ne sont pas réduits en $\text{CCI}_\square$, ce qui motive la définition suivante :

Définition 3.2 (opération `wctxt`)**.** Soit $\to$ une règle de réduction. La *clôture par contexte faible* $\to_{\mathcal{W}}$ est définie comme `ctxt` (définition 2.6), en enlevant les règles `Ctx_lam_r` et `Ctx_prd_r`.

De plus, si on regarde la définition 2.9, la règle `iota_case` ne peut plus convenir lorsque l'objet filtré est logique (*i.e.* de sorte `Prop`) : en effet, l'objet filtré peut devenir $\square$ après extraction. Cependant,

pour qu'un filtrage sur un inductif logique intervienne dans un « véritable » calcul, il faut qu'il ait exactement un constructeur dont l'argument est purement logique, en vertu de la remarque 2.8. Un problème similaire se présente avec les points fixes dont l'argument de décroissance est logique (`iota_fix`). C'est pourquoi nous introduisons les deux règles de réduction `dummy_fix` et `dummy_case` :

Définition 3.3 (prédicats `dummy_redn_term` et $\rightarrow_\Box$)**.** La règle `dummy_redn_term` est formée de l'union de `redn_term` (définition 2.9) et des règles suivantes :

$$\frac{}{\texttt{Fix}(f:T)\ p\ \Box \rightarrow_{\Box_f} f\ (\texttt{Fix}(f:T))\ p\ \Box}\ \texttt{dummy_fix}$$

$$\frac{}{\texttt{Case}\ \{\vec{p}\}\ (\Box, P, \vec{f}) \rightarrow_{\Box_c} f_1\ \Box}\ \texttt{dummy_case}$$

On notera $\rightarrow_\Box$ la clôture par contexte faible de `dummy_redn_term`.

Petit point sur les notations « $\Gamma \vdash t \rightarrow_\Box u$ » désigne ce que l'on note en Coq « `wctxt dummy_redn_term` $\Gamma\ t\ u$ ». Il s'agit de *la* règle correspondant à une étape de réduction du CCI$_\Box$.

3.3. Propriétés de $\rightarrow_\mathcal{E}$

Cette section est une application assez fidèle de l'étude syntaxique menée par P. Letouzey dans la section 2.3 de [12] au CCI de B. Barras.

Nous allons énoncer des propriétés de $\rightarrow_\mathcal{E}$ qui ont été prouvées en Coq dans le cadre du travail présenté ici. Les présentations des lemmes ci-dessous feront appel à des notions et notations qui ne seront pas définies formellement ici, mais juste expliquées informellement. Bien sûr, ces notions ont été formellement définies en Coq.

Lemme 3.4 (`Pe_weak`)**.** *La règle suivante est admissible :*

$$\frac{\Gamma_1\Gamma_2 \vdash t \rightarrow_\mathcal{E} t' \qquad \Gamma_1\delta \vdash}{\Gamma_1\delta\Gamma_2' \vdash \uparrow^1_{|\Gamma_2|} t \rightarrow_\mathcal{E} \uparrow^1_{|\Gamma_2|} t'}$$

Cette règle dit que l'extraction est stable par ajout d'une déclaration δ dans un environnement $\Gamma_1\Gamma_2$. Γ_2' représente l'environnement Γ_2 ayant subi les opérations idoines de relocation. On remarquera la ressemblance entre ce lemme et le lemme d'*affaiblissement* que l'on rencontre en typage [5]. Le script de preuve en Coq est d'ailleurs quasiment identique.

Le lemme suivant est aussi similaire à un lemme de typage, le lemme de *substitution* [6] :

Lemme 3.5 (`Pe_sub`)**.** *La règle suivante est admissible :*

$$\frac{\Gamma_1[x:U] \vdash t:T \qquad \Gamma_2 \vdash u:U \qquad t \rightarrow_\mathcal{E} t' \qquad u \rightarrow_\mathcal{E} u'}{t\,\{\natural 0 \leftarrow u\} \rightarrow_\mathcal{E} t'\,\{\natural 0 \leftarrow u'\}}$$

Le théorème suivant est un des principaux résultats de [12] — il s'agit du théorème 2. Il exprime que l'on peut simuler au niveau CCI toute réduction d'un terme extrait.

Théorème 3.6 (`Pe_wctxt_correct`)**.** *Soient t un terme CCI clos (sans variable libre, ni constante globale) bien typé et t', u' deux termes de CCI$_\Box$ tels que $t \rightarrow_\mathcal{E} t'$ et $t' \rightarrow_\Box u'$. Il existe alors un terme CCI u tel que $u \rightarrow_\mathcal{E} u'$ et $t \rightarrow_\mathcal{C}^+ u$.*

5. lemme 5.25 `typ_weak` de [1], page 140
6. lemme 5.26 `typ_sub` de [1], page 140

$$\begin{tikzcd} t \arrow[r, dashed, "\rightarrow_c^+"] \arrow[d, "\rightarrow_\mathcal{E}"'] & u \arrow[d, dashed, "\rightarrow_\mathcal{E}"] \\ t' \arrow[r, "\rightarrow_\square"'] & u' \end{tikzcd}$$

En Coq, ce théorème de relèvement s'exprime sous la forme suivante :

```
Theorem Pe_wctxt_correct : forall e t t' u' T,
  closed O t ->
  gclosed t ->
  sg -- e |- t : T ->
  Pe e t t' ->
  wctxt dummy_redn_term e t' u' ->
  exists u, R_t (wctxt redn_term) e t u /\ Pe e u u'.
```

La preuve suit le même principe que dans [12] : on procède par cas selon la réduction employée entre t' et u'. Pour chaque redex dans t', on trouve un redex correspondant dans t. La preuve en Coq est cependant plus longue et pénible par moments, notamment à cause de la représentation des termes avec opérateurs qui laisse la possibilité de nombreux termes absurdes (tels que l'opérateur @ appliqué à autre chose qu'un couple). Pour éliminer ces termes mal formés, nous avons utilisé le typage, alors que l'on pourrait s'attendre à n'utiliser le typage que pour traiter la règle `Pe_Prop`. Il est important de noter que les termes extraits ne sont pas typés : le fait que les termes extraits sont issus de termes bien typés est alors crucial pour justifier leur « bonne forme ».

Dans l'hypothèse où le CCI normalise fortement, on prouve facilement la normalisation forte des termes extraits.

Corollaire 3.7 (`Pe_redn_sn`). *Soit t un terme clos bien typé dans CCI et t' tel que $t \rightarrow_\mathcal{E} t'$. Alors toute suite de dérivations $\rightarrow_\square$ partant de t' est finie.*

Finalement, nous avons montré qu'un calcul sur un terme extrait termine toujours sur une réponse reliée au terme CCI correspondant. Par contre, nous n'avons pas encore prouvé que le calcul ne bloque pas trop tôt, sur un terme non totalement réduit vers une valeur.

3.4. Une extraction extraite

B. Barras a réalisé un prototype de système de preuves dans lequel le typeur est obtenu via la « vraie » extraction de Coq, auquel est ajoutée une interface sommaire écrite en OCaml. Nous avons ajouté à ce système une commande d'extraction en implémentant une fonction `extract_term` vérifiant pour tout terme t clos bien typé la propriété :

$$t \rightarrow_\mathcal{E} \texttt{extract_term}(t)$$

Cette fonction est essentiellement une déterminisation de la relation $\rightarrow_\mathcal{E}$ privilégiant les règles d'élagage, afin d'obtenir l'extraction la plus précise possible.

À titre d'illustration, voici une session de l'interpréteur mettant en œuvre l'extraction :

```
Bcoq < Inductive nat: Set := O: nat | S: (!nat _ ~)->nat.
nat defini(s) inductivement.

Bcoq < Definition plus: nat -> nat -> nat :=
```

```
  (Fix {X, F | (nat-X)->nat->nat :> nat->nat->nat :=
       [n:nat+X]<[a:Lmark][_:(!nat ~ a)]nat->nat>Cases n of
          | O => [_:Lmark][m:nat]m
          | S => [pn:(nat-X)*Lmark][m:nat](S (F^0 ~ pn^0 m))
          end}^0 ~).
plus defini.

Bcoq < Print plus.
plus:
(P1 Fix((Lmark->nat->nat->nat)*Lmark,[X:Mark]
  [F:(Lmark->Ind{nat;0}(~,(~,::(X,~)))->nat->nat)*Lmark]
  ([_:Lmark][n:Ind{nat;1}(~,(~,::(X,~)))]Case{O|S}(n,([a:Lmark]
    ([_0:Ind{nat;0}(~,(a,~))]nat->nat,([_0:Lmark][m:nat]m,
      ([pn:Ind{nat;0}(~,(~,::(X,~)))*Lmark][m:nat]
        Cstr{S}(~,(P1 F ~ P1 pn m,~)),~)))),~)) ~)

Bcoq < Extraction plus.
Extraction de plus:
(P1 Fix(#,[X:#]
  [F:#]
  ([_:#][n:#]Case{O|S}(n,(#,
    ([_0:#][m:#]m,
      ([pn:#][m:#]
        Cstr{S}(#,(P1 F # P1 pn m,#)),#)))),#)) #)
```

L'invite de l'interpréteur est `Bcoq <`, et chaque commande entrée par l'utilisateur se termine par un point. Dans cette session, le type des entiers est défini, puis l'addition sur les entiers. La syntaxe est encore lourde, et nous avons rajouté manuellement des sauts de lignes et l'indentation sur les sorties de `Print plus` et de `Extraction plus` afin de les faire correspondre.

4. Conclusion, perspectives

Nous avons présenté une formalisation de l'extraction dans le cadre de Coq-en-Coq. Nous n'avons traité qu'une extraction très simple, produisant des termes généralement verbeux et ne gérant pas les constantes. L'optimisation des programmes extraits — en éliminant *complètement* les □ inutiles — n'a pas encore été abordée.

Cela n'est qu'une manière d'aborder la garantie formelle pour l'extraction Coq. D'autres travaux (en cours) visent à concevoir une extraction pour le « vrai » sytème Coq, générant des preuves de correction en même temps que les programmes extraits.

Enfin, nous avons occulté l'exécution par de vraies machines de nos programmes extraits. C'est le rôle des interpréteurs et des compilateurs. Notre but ultime est la compilation en programme machine. Cette phase est complexe, en particulier pour les langages fonctionnels, qui offrent un très haut niveau d'abstraction. Le développement d'un compilateur de mini-ML certifié est en cours [5], ce qui, combiné à nos travaux, permettra d'avoir une chaîne complète de certification depuis la spécification d'une fonction en Coq jusqu'à son exécution sur un processeur PowerPC [11].

Références

[1] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, November 1999.

[2] S. Berghofer. A constructive proof of Higman's lemma in Isabelle. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[3] S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic.* PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[4] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[5] Z. Dargaye. Décurryfication certifiée. In *Journées Françaises sur les Langages Applicatifs JFLA'07*, 2007.

[6] H. Benl et al. Proof theory at work : Program development in the Minlog system. In Wolfgang Bibel and Peter H. Schmidt, editors, *Automated Deduction : A Basis for Applications. Volume II, Systems and Implementation Techniques.* Kluwer Academic Publishers, Dordrecht, 1998.

[7] H. Geuvers. *Inconsistency of classical logic in type theory.* Short note, 2001.

[8] W.A. Howard. The formulae-as-types notion of constructions. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980. Unpublished 1969 Manuscript.

[9] S. C. Kleene. *Introduction to Metamathematics.* North-Holland, Amsterdam, 1952.

[10] C. Kreitz. *The Nuprl Proof Development System, Version 5.* Cornell University, Ithaca, NY, 2002. Available at `http://www.nuprl.org`.

[11] X. Leroy. Formal certification of a compiler back-end or : programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006.

[12] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris-Sud, July 2004.

[13] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM Press.

[14] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse d'université, Paris 7, January 1989.

[15] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15 :607–640, 1993.

Abstraction d'horloges dans les systèmes synchrones flot de données *

Louis Mandel & Florence Plateau

LRI, Université Paris-Sud 11
INRIA Saclay
{mandel,plateau}@lri.fr

Résumé

Les langages synchrones flot de données tels que Lustre manipulent des séquences infinies de données comme valeurs de base. Chaque flot est associé à une *horloge* qui définit les instants où sa valeur est présente. Cette horloge est une information de type et un système de types dédié, le calcul d'horloges, rejette statiquement les programmes qui ne peuvent pas être exécutés de manière synchrone. Dans les langages synchrones existants, cela revient à se demander si deux flots ont la même horloge et repose donc uniquement sur l'égalité d'horloges. Des travaux récents ont montré l'intérêt d'introduire une notion relâchée du synchronisme, où deux flots peuvent être composés dès qu'ils peuvent être synchronisés par l'introduction d'un buffer de taille bornée (comme c'est fait dans le modèle SDF d'Edward Lee). Techniquement, cela consiste à remplacer le typage par du sous-typage. Ce papier est une traduction et amélioration technique de [11] qui présente un moyen simple de mettre en oeuvre ce modèle relâché par l'utilisation d'horloges abstraites. Les valeurs abstraites représentent des ensembles d'horloges concrètes qui ne sont pas nécessairement périodiques. Cela permet de modéliser divers aspects des logiciels temps-réel embarqués, tels que la gigue bornée présente dans les systèmes vidéo, le temps d'exécution des processus temps réel et, plus généralement, la communication à travers des buffers de taille bornée. Nous présentons ici l'algèbre des horloges abstraites et leur principales propriétés théoriques.

1. Introduction

Les langages synchrones flot de données tels que Lustre [3] ont été introduits dans les années 80 pour l'implémentation de systèmes temps-réel critiques. Depuis, ils ont été utilisés dans diverses applications industrielles telles que les commandes de vol dans les avions Airbus. Ils ont été créés dans l'objectif de construire un langage de programmation proche des modèles mathématiques utilisés dans les systèmes embarqués tels que les équations sur des flots de données ou la composition d'automates à états finis. Dans ces langages, le synchronisme a une justification très pratique : à un certain niveau d'observation, le temps est considéré comme une séquence de réactions instantanées du système à des événements extérieurs et quand les processus sont composés en parallèle, ils s'accordent sur ces réactions [4]. Cela coïncide aussi avec l'interprétation de Milner du synchronisme dans SCCS [19] ou avec le produit synchronisé d'automates [2]. Dans les langages flot de données comme Lustre, le synchronisme est celui de la théorie du contrôle et peut-être interprété comme une contrainte de typage sur les séquences : quand on combine deux séquences $(x_i)_{i\in D_1}$ et $(y_i)_{i\in D_2}$ dans $(x_i)_{i\in D_1} + (y_i)_{i\in D_2}$, leurs domaines de temps D_1 et D_2 doivent être compatibles. Cette vérification statique est réalisée par un système de types appelé *calcul d'horloges* [12, 1] qui impose que D_1 et D_2 soient égaux. Une telle analyse n'est pas bornée aux langages synchrones et est d'un intérêt plus large.

*. Ce travail a été partiellement soutenu par le projet de recherche INRIA Synchronics.

Par exemple, une analyse des horloges est faite dans les outils de modélisation comme SIMULINK [9] ou SCICOS [21]. Le calcul d'horloges consiste essentiellement à se demander si deux flots ont la même horloge. Pour cela, les informations d'horloges sont représentées par des types. Considérons le langage de types suivant (tiré de [12]) :

schémas d'horloges	σ	$::=$	$\forall\alpha.\ \forall X_1, ..., X_m.\ ct$
types d'horloges	ct	$::=$	$ct \to ct \mid ct * ct \mid ck \mid (X : ck)$
horloges	ck	$::=$	ck `on` $c \mid ck$ `on` ***not*** $c \mid \alpha$
séquences booléennes	c	$::=$	$X \mid n$ où n est un ensemble de noms dénombrable

Comme dans le système de types de Hindley-Milner [18], les types sont séparés en schémas d'horloges (σ) et types d'horloges (ct). Un type d'horloges est quantifié sur des variables d'horloges (α) ou des variables booléennes (X). Le type d'horloges de ($+$) est $\forall\alpha.\ \alpha \times \alpha \to \alpha$, qui indique que si x et y sont deux flots de même horloge α alors $x + y$ a aussi l'horloge α. Exprimé différemment, l'addition attend deux arguments synchrones et produit un flot résultat sur la même horloge. Un autre exemple de primitive synchrone est le délai unitaire, qui décale son entrée. Si x et y sont deux séquences $x_1\, x_2\, x_3 ...$ et $y_1\, y_2\, y_3 ...$ alors x `fby` y représente $x_1\, y_1\, y_2\, y_3 ....$ x et y doivent avoir la même horloge et x `fby` y est produit sur cette horloge. On peut donner à `fby` la signature d'horloges : $\forall\alpha.\ \alpha \times \alpha \to \alpha$.

Les choses deviennent plus intéressantes quand l'échantillonnage intervient. Deux structures de contrôle typiques sont les structures d'échantillonnage et de combinaison de flots :

$$\begin{array}{lcl} \texttt{when} & : & \forall\alpha.\ \forall X.\ \alpha \to (X : \alpha) \to \alpha\ \texttt{on}\ X \\ \texttt{merge} & : & \forall\alpha.\ \forall X.\ (X : \alpha) \to \alpha\ \texttt{on}\ X \to \alpha\ \texttt{on}\ \textit{not}\ X \to \alpha \end{array}$$

x `when` y est bien typé pour le calcul d'horloges si x et y ont la même horloge α. Dans ce cas, le résultat a une horloge plus lente correspondant aux instants où y est vrai et on écrit cela α `on` y (la meta-variable X est substituée par la valeur effective y). Par exemple, si *half* est une séquence booléenne 10101010101... (c'est à dire l'expression régulière $(10)^\omega$, notée par la suite (10)) et x est une séquence $x_1\, x_2\, x_3 ...$ alors x `when` *half* est un échantillonnage de x de fréquence un demi, c'est à dire $x_1\, x_3\, x_5 ....$ Si x a un type d'horloges ck, alors x `when` *half* a le type d'horloges ck `on` *half*. L'expression $x + (x$ `when` *half*$)$, qui calculerait la séquence $(x_i + x_{2i})_{i \in \mathbb{N}}$ n'est pas bien typée puisque ck n'est pas égal à ck `on` *half*, et elle est statiquement rejetée par le compilateur. L'opérateur `merge` est symétrique : il attend une horloge, une séquence sur cette horloge et une séquence sur le complémentaire de cette horloge et les combine pour construire une séquence plus longue.

Pour comparer les horloges, la plupart des implémentations s'en tiennent à l'égalité des noms : ck `on` n_1 et ck `on` n_2 peuvent être unifiées seulement si $n_1 = n_2$. Cette restriction forte est raisonnable quand n_1 est le résultat d'un calcul complexe (pour lequel l'égalité est non décidable). Elle est néanmoins trop restrictive pour un ensemble important d'applications où les horloges sont périodiques, car elle ne permet pas de tenir compte des propriétés de ces horloges. En particulier, des travaux récents ont montré l'intérêt d'un modèle relâché de synchronisme, permettant de composer des flots dès qu'ils peuvent être synchronisés par insertion de buffers bornés. Ce modèle est appelé modèle de Kahn n-synchrone [10] et poursuit les travaux fondateurs de Edward Lee sur les *Synchronous Data-Flow graphs* (SDF) [17, 8].

Du point de vue du typage, le n-synchronisme revient à rajouter une règle de sous-typage au système de types qu'est le calcul d'horloges :

$$\text{(SUB)}\quad \frac{H \vdash e : ck \quad ck <: ck'}{H \vdash e : ck'}$$

Intuitivement, $ck <: ck'$ assure qu'un flot produit sur l'horloge ck peut être consommée sur ck' par insertion d'un buffer borné. Dans le cas particulier des *horloges ultimement périodiques* [25], le sous-typage est décidable. Les détails techniques sont rappelés en section 2.

Le point de vue adopté dans [11] est légèrement différent et plus simple que celui adopté dans [10]. Au lieu de se concentrer sur les horloges périodiques, on donne la possibilité de raisonner sur des ensembles d'horloges comme des abstractions [13] des horloges concrètes. En effet, dans certaines applications, les horloges exactement périodiques ne sont pas indispensables et il est suffisant de raisonner sur des intervalles d'horloges dont les bornes sont néanmoins périodiques. C'est typiquement le cas dans quatre genres d'applications nécessitant la communication par buffers (1) les applications avec gigue bornée, (2) les applications utilisant des horloges dont la taille du motif périodique rend rédhibitoire les calculs exacts, (3) les applications nécessitant la modélisation du temps d'exécution des processus temps-réel (4) les applications où les noeuds de calcul manipulent plusieurs valeurs d'un même flot au sein d'un instant logique.

Par exemple, un flot x qui est présent en moyenne 3 fois sur 7 par rapport à une horloge de base ck et est soumis à une potentielle gigue de 4 instants aura le type d'horloges $\exists n \in [-2,2](7/3).\ ck\ \texttt{on}\ n$. Le quantificateur existentiel cache l'instant exact où l'élément est présent mais en donne une borne. Intuitivement, la notation $[-2,2](7/3)$ représente toutes les horloges dont le $j^{\text{ème}}$ 1 se trouve de deux instants avant à deux instants après le $j^{\text{ème}}$ 1 d'une « horloge parfaite » dont les occurrences de 1 seraient séparés par exactement sept tiers d'instant. Si f est de la forme $\lambda x.x\ \texttt{when}\ e$ pour une certaine expression booléenne compliquée e mais dont on peut prouver qu'elle appartient à l'ensemble des valeurs représentées par $[-2,2](7/3)$, alors un type d'horloges valide pour f est $\forall \alpha.\ \alpha \to \alpha\ \texttt{on}\ n$ avec $n \in [-2,2](7/3)$.

Le concept et une mise en oeuvre de l'abstraction d'horloges ont été présentés dans [11]. Cet article reprend en français la structure du précédent, en le résumant et en introduisant une nouvelle abstraction plus simple et plus précise.

Après la définition formelle des mots binaires infinis et de leur propriétés utiles au modèle n-synchrone (section 2), nous présenterons la nouvelle version de l'abstraction et ses propriétés algébriques (section 3). Nous discuterons ensuite de ses avantages par rapport à la précédente (section 4), avant de présenter des applications du mécanisme d'abstraction d'horloges (section 5).

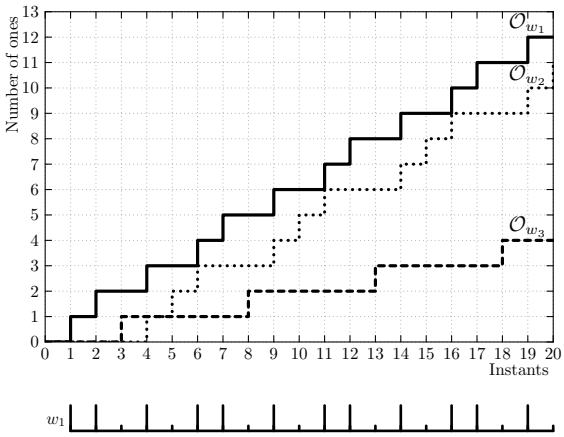
Un grand nombre de propositions énoncées dans ce papier ont été prouvées en Coq [6]. La formalisation et la preuve en Coq ont été un support à la réflexion, et nous ont surtout permis d'essayer plusieurs mécanismes d'abstraction différents. La modification de l'abstraction nécessite l'adaptation des preuves. Cette activité source d'erreurs à la main était dans notre cas automatiquement vérifiée par le compilateur Coq. Cela nous a permis d'avoir une plus grande confiance en les différents mécanismes essayés, et en particulier en celui que nous avons retenu.

Une version longue de l'article ainsi que les développements Coq sont disponibles à l'adresse http://www.lri.fr/~plateau/jfla09. Les ✿ pointent vers le code Coq correspondant à la définition ou la propriété énoncée.

2. Horloges et mots binaires infinis

Une relation de sous-typage peut-être vérifiée seulement si les types d'horloges sont exprimés par rapport à la même variable d'horloge. Intuitivement, cela tient au fait que l'échantillonnage est toujours relatif à une horloge ($ck\ \texttt{on}\ c$ est relatif à ck). Dans ce cas la relation de sous-typage correspond à une relation sur les séquences booléennes $\alpha\ \texttt{on}\ c <: \alpha\ \texttt{on}\ c' \Leftrightarrow c <: c'$.

Dans cette section, nous présentons les mots binaires infinis et une opération booléenne $\boldsymbol{on}$ sur celles-ci, telle que $(ck\ \texttt{on}\ c_1)\ \texttt{on}\ c_2 = ck\ \texttt{on}\ (c_1\ \boldsymbol{on}\ c_2)$. Nous présentons ensuite la relation de sous-typage sur ces mots. Comme nous manipulons principalement la partie « flot booléen » (c) du type d'horloges, nous l'appellerons aussi horloge.



Les chronogrammes représentent les fonctions $\mathcal{O}_w$, qui associent à chaque instant i le nombre de 1 vus dans w depuis l'instant 1. Un front montant à l'instant i signifie que l'élément à l'indice i de w est un 1 (*i.e.* $w[i] = 1$). Si la fonction ne change pas de valeur à l'instant i, alors l'élément à l'indice i de w est un 0 (*i.e.* $w[i] = 0$).
Le chronogramme à une dimension situé sous le graphique représente le mot w_1, c'est une projection de $\mathcal{O}_{w_1}$.

FIGURE 1 – Chronogrammes représentant les mots $w_1 = (11010)$, $w_2 = 0(00111)$, $w_3 = (00100)$.

2.1. Définitions

Une horloge peut-être un mot binaire infini ou une composition de ceux-ci. Si on identifie les noms à leurs valeurs, les horloges ont la grammaire suivante :

$$\begin{array}{lcl} c & ::= & w \mid \textbf{\textit{not}}\ w \mid c\ \textbf{\textit{on}}\ c \\ w & ::= & 0.w \mid 1.w \end{array}$$

$\textbf{\textit{not}}\ w$ est la négation de w, $c_1\ \textbf{\textit{on}}\ c_2$ est l'horloge échantillonnée et w un mot binaire infini. Dans les mots binaires infinis, un 1 dénote la présence d'une valeur sur le flot et un 0 l'absence de valeur.

La concaténation d'un mot binaire fini u et d'un mot binaire v est notée $u.v$. On notera 0^n la concaténation de n valeurs 0, 1^n la concaténation de n valeurs 1. On appellera *mots binaires infinis ultimement périodiques*, ou plus simplement *mots binaires périodiques* et on notera $u(v)$, les mots constitués d'un préfixe fini u suivi de la répétition infinie d'un mot fini v. L'élément à l'indice i de w est noté $w[i]$ (le premier élément du mot est à l'indice 1). On définit la fonction $\mathcal{O}_w(i)$ qui donne le nombre de 1 apparaissant dans w jusqu'à l'indice i :

Définition 1 (nombre de 1 vus dans w jusqu'à l'indice i : $\mathcal{O}_w$)**.**

$$\begin{array}{lcl} \mathcal{O}_w(0) & = & 0 \\ \forall i \geq 1,\ \mathcal{O}_w(i) & = & \begin{cases} \mathcal{O}_w(i-1) & si\ w[i] = 0 \\ \mathcal{O}_w(i-1) + 1 & si\ w[i] = 1 \end{cases} \end{array}$$

Par convention, le nombre de 1 vus dans w avant l'instant 1 vaut 0.

La fonction $\mathcal{O}_w$ est une fonction discrète qui ne varie que par fronts montants de hauteur 1 ($\forall i \geq 0,\ 0 \leq \mathcal{O}_w(i+1) - \mathcal{O}_w(i) \leq 1$).

Notons que l'on peut toujours reconstruire un mot w à partir de sa fonction $\mathcal{O}_w$.

La figure 1 montre des exemples de mots binaires infinis, représentés par les chronogrammes de leur fonction $\mathcal{O}_w$. Si un flot est produit (resp. consommé) sur l'horloge w, alors une valeur est produite (resp. consommée) à chaque front montant du chronogramme.

Définissons maintenant l'opérateur d'échantillonnage. Si un flot x de type d'horloges α `on` w_1 est échantillonné avec l'horloge w_2, alors le flot résultant $x' = x$ `when` w_2 sera de type d'horloges α `on` $(w_1\ \textbf{\textit{on}}\ w_2)$. Si x est présent, alors w_2 est consulté pour savoir si la valeur doit être conservée. Si x est absent, alors x' est lui aussi absent et w_2 n'est pas consulté.

Définition 2 (opérateur *on*). ✿

$$\begin{array}{lcl} 1.w_1 \ \textit{on}\ 1.w_2 & = & 1.(w_1 \ \textit{on}\ w_2) \\ 1.w_1 \ \textit{on}\ 0.w_2 & = & 0.(w_1 \ \textit{on}\ w_2) \\ 0.w_1 \ \textit{on}\ w_2 & = & 0.(w_1 \ \textit{on}\ w_2) \end{array}$$

Par exemple (10) *on* (10) est égal à (1000).

Les éléments de w_1 *on* w_2 correspondent aux éléments de w_2, quand w_2 est parcouru au rythme des 1 dans w_1. Donc on sait que si le nombre de 1 vus dans w_1 à l'indice i est j, alors le nombre de 1 vus dans w_1 *on* w_2 à l'indice i est égal au nombre de 1 vus dans w_2 à l'indice j :

$$\forall i \geq 0,\ \mathcal{O}_{w_1 \ \textit{on}\ w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i)) \quad ✿$$

Cette formulation de l'opérateur *on* est très utile pour les preuves. Par exemple, il est alors simple de prouver que l'opérateur *on* est associatif : $(w_1 \ \textit{on}\ w_2) \ \textit{on}\ w_3 = w_1 \ \textit{on}\ (w_2 \ \textit{on}\ w_3)$. ✿

2.2. Taille de buffer

La taille de buffer minimale pour transmettre un flot produit sur une sortie de type d'horloges α `on` w vers une entrée de type d'horloges α `on` w' est la différence maximale entre le nombre de valeurs produites et le nombre de valeurs consommées au cours de l'exécution :

$$size(w, w') = \max_{i \in \mathbb{N}}(\mathcal{O}_w(i) - \mathcal{O}_{w'}(i))$$

Notons que si le minimum de cette différence est négatif, alors au moins une lecture dans le buffer se produira à un instant où celui-ci est vide. En effet, quand une valeur négative est atteinte, moins de valeurs ont été produites que consommées.

Dans les graphiques, la taille de buffer nécessaire pour communiquer de α `on` w vers α `on` w' est égale à la plus grande différence entre les courbes $\mathcal{O}_w$ et $\mathcal{O}_{w'}$. Par exemple, dans la figure 1, le nombre maximum de données produites et pas encore consommées durant la communication de α `on` w_1 vers α `on` w_2 est 2, atteint pour la première fois à l'instant 1. En revanche, le nombre de données à stocker durant la communication de α `on` w_1 vers α `on` w_3 augmente à l'infini.

2.3. Relation de sous-typage

La relation de sous-typage α `on` $w_1 <: \alpha$ `on` w_2 est vérifiée si et seulement si un flot produit sur α `on` w_1 peut être consommé sur α `on` w_2 par insertion d'un buffer de taille bornée.

Pour cela, il faut que les données soient toujours produites avant d'être attendues (relation de *précédence*), et que le nombre de valeurs présentes dans le buffer au cours de l'exécution soit borné (relation de *synchronisabilité*).

Définition 3 (précédence $\preceq$). ✿
On dit que w_1 précède w_2 et on note $w_1 \preceq w_2$ si et seulement si $\forall i \geq 0,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$.

Un mot w_1 précède un mot w_2 si à tout instant il a produit autant ou plus de 1 que w_2. Cette relation est un ordre partiel. ✿ Elle permet de vérifier que la relation de causalité entre les flots est préservée, c'est à dire que le producteur écrit toujours ses sorties dans le buffer avant que le consommateur n'en aie besoin.

Sur la figure 1, $\mathcal{O}_{w_1}$ est toujours au dessus de $\mathcal{O}_{w_2}$ et $\mathcal{O}_{w_3}$, donc $w_1 \preceq w_2$ et $w_1 \preceq w_3$. Par contre, $\mathcal{O}_{w_2}$ et $\mathcal{O}_{w_3}$ sont entrelacées, donc $w_2 \not\preceq w_3$ et $w_3 \not\preceq w_2$.

Le supremum $\sqcup$ et l'infimum $\sqcap$ d'un ensemble de mots binaires infinis $W = \{w_1, \ldots, w_n\}$ pour la relation de précédence sont définis par :

$$\forall i \geq 0,\ \mathcal{O}_{\sqcup W}(i) = \min_{w \in W}(\mathcal{O}_w(i)) \quad ✿$$
$$\forall i \geq 0,\ \mathcal{O}_{\sqcap W}(i) = \max_{w \in W}(\mathcal{O}_w(i)) \quad ✿$$

Sur la figure 1, $w_2 \sqcup w_3$ est égal à w_2 jusqu'à l'instant 4, et ensuite égal à w_3. Les fronts montants du supremum sont situés à l'indice maximal des fronts montants des mots considérés. Pour tout $w \in W$, $\sqcap W \preceq w \preceq \sqcup W$. ✿ ✿

Définition 4 (synchronisabilité $\bowtie$)**.** ✿ *On dit que w_1 et w_2 sont synchronisables et on note $w_1 \bowtie w_2$ si et seulement si $\exists b_1, b_2 \in \mathbb{Z},\ \forall i \geq 0,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$.*

Deux mots w_1 et w_2 sont synchronisables si la différence entre le nombre de 1 vus dans w_1 et le nombre de 1 vus dans w_2 est bornée. [1] Cette relation est une relation d'équivalence. ✿

Elle permet de vérifier qu'il existe une taille correcte pour le buffer, c'est à dire telle qu'il n'y aura jamais de dépassement de capacité.

Par exemple, sur la figure 1, $\mathcal{O}_{w_1}$ et $\mathcal{O}_{w_2}$ restent à une distance verticale bornée l'une de l'autre, donc $w_1 \bowtie w_2$ ($\forall i \geq 0,\ 0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq 2$). Un buffer de taille 2 permettra une communication sans perte. En revanche, la courbe de w_3 est de plus en plus basse par rapport à celles de w_1 et w_2, donc $w_3 \not\bowtie w_1$ et $w_3 \not\bowtie w_2$ ($\forall i \geq 0,\ -\infty \leq \mathcal{O}_{w_3}(i) - \mathcal{O}_{w_2}(i) \leq 1$ et $-\infty \leq \mathcal{O}_{w_3}(i) - \mathcal{O}_{w_1}(i) \leq 0$).

Définition 5 (sous-typage $<:$)**.** ✿ *On dit que w_1 est un sous-type de w_2 et on note $w_1 <: w_2$ si et seulement si $w_1 \preceq w_2$ et $w_1 \bowtie w_2$.*

En effet, si les données sont toujours écrites avant d'être lues, et si le nombre de données en attente de lecture ne croit pas à l'infini, alors la communication peut-être rendue synchrone par l'insertion d'un buffer de taille bornée. Par exemple, dans la figure 1, $w_1 <: w_2$.

Toutes les définitions de cette section peuvent être étendues aux horloges composées (c) par calcul des opérateurs ***not*** et ***on***.

3. Abstraction des horloges

L'abstraction d'un mot binaire infini w que nous proposons ici consiste à conserver uniquement le taux asymptotique r de 1 dans w et deux valeurs qui donnent les décalages minimum et maximum du nombre de 1 vus dans w par rapport au taux r. On note cette valeur abstraite $\left(b^0, b^1, r\right)$.

Les horloges abstraites peuvent être une valeur abstraite ou une composition de valeurs abstraites. Elles sont définies par la grammaire suivante : [2]

$$\begin{array}{lcll} ac & ::= & a \mid \textbf{\textit{not}}^{\sim}\ a \mid ac\ \textbf{\textit{on}}^{\sim}\ ac & ✿ \\ a & ::= & \left(b^0, b^1, r\right) \text{ avec } b^0, b^1, r \in \mathbb{Q}, 0 \leq r \leq 1 & ✿\ ✿ \end{array}$$

Dans cette section, nous allons décrire l'ensemble d'horloges représenté par une valeur abstraite et nous montrerons que cet ensemble est reconnaissable par un automate fini. Les opérateurs $\textbf{\textit{on}}^{\sim}$ et $\textbf{\textit{not}}^{\sim}$ sont ensuite définis. Ils permettent de réduire de manière efficace une horloge abstraite en une valeur abstraite. Finalement, nous montrerons que les relations présentées en section 2 peuvent être facilement vérifiées sur une abstraction, et qu'une taille de buffer correcte peut être calculée de manière efficace.

1. Cette définition de la relation de synchronisabilité est moins restrictive que celle de [10], mais correspond exactement à ce que l'on veut exprimer ici : si $w_1 \bowtie w_2$, la taille de buffer nécessaire pour communiquer de w_1 à w_2 ou de w_2 à w_1 est bornée.
2. $\mathbb{Q}$ est l'ensemble des nombres rationnels.

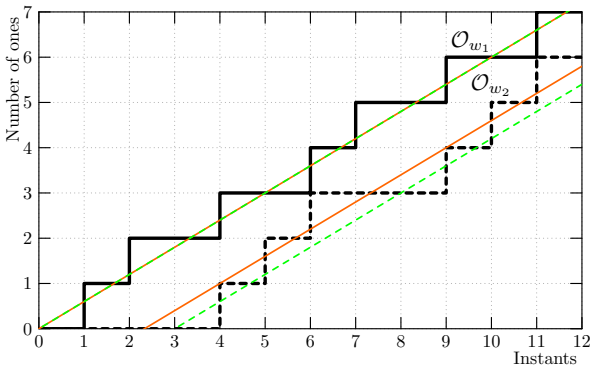


FIGURE 2 – Si les fronts montants d'un mot w débutent tous sous la droite Δ^1 d'équation $r \times i + b^1$ et l'absence de front ne se produit qu'au dessus de la droite Δ^0 d'équation $r \times i + b^0$, alors w est dans l'abstraction $\left(b^0, b^1, r\right)$. Donc w_1 est dans $a_1 = \left(0, 0, \frac{3}{5}\right)$ et w_2 est dans $a_2 = \left(-\frac{9}{5}, -\frac{7}{5}, \frac{3}{5}\right)$.

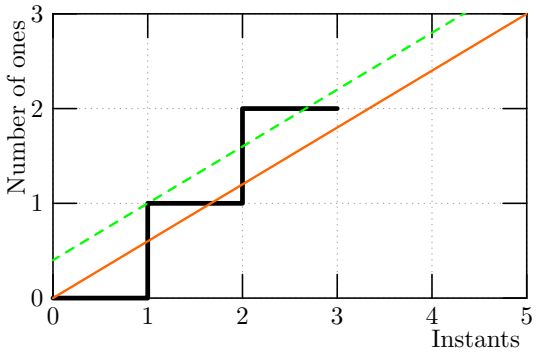


FIGURE 3 – La concrétisation de $a_3 = \left(\frac{2}{5}, 0, \frac{3}{5}\right)$ est vide : il n'y a pas d'élément valide au troisième instant. En effet, la courbe se situe au dessus de la droite Δ^1, donc un front montant n'est pas autorisé, et en dessous de la droite Δ^0 donc une absence de front n'est pas autorisée non plus.

3.1. Abstraction des mots binaires infinis

Une valeur abstraite $\left(b^0, b^1, r\right)$ représente l'ensemble de mots binaires infinis suivant :

Définition 6 (concrétisation). ✿

$$concr\left(\left(b^0, b^1, r\right)\right) \overset{def}{\Leftrightarrow} \left\{ w,\ \forall i \geq 1, \quad \begin{array}{lcl} w[i] = 1 & \Rightarrow & \mathcal{O}_w(i-1) < r \times i + b^1 \\ \wedge & & \\ w[i] = 0 & \Rightarrow & \mathcal{O}_w(i-1) \geq r \times i + b^0 \end{array} \right\}$$

Un taux $r > 1$ représenterait des mots qui ont en moyenne plus qu'un 1 par instant, qui ne sont pas considérés ici (comme il est mentionné en section 2.1, $\mathcal{O}_w(i+1) - \mathcal{O}_w(i) \leq 1$).

Dans les chronogrammes, l'abstraction $\left(b^0, b^1, r\right)$ peut être représentée par deux droites Δ^0 et Δ^1 qui bornent les fronts montants et les absences de fronts des mots qu'elle contient. Tout front montant doit démarrer sous la droite Δ^1 (représentée par une ligne rouge continue) et toute absence de front démarre au dessus de la droite Δ^0 (représentée une ligne verte en pointillés). Les équations de ces droites sont $\Delta^1 :\ r \times i + b^1$ et $\Delta^0 :\ r \times i + b^0$. Par exemple, dans la figure 2 le mot $w_2 = 0(00111)$ est abstrait par $a_2 = \left(-\frac{9}{5}, -\frac{7}{5}, \frac{3}{5}\right)$. Le mot w_1 est abstrait par a_1 dont les droites Δ^0 et Δ^1 sont confondues.

On peut remarquer que tout mot restant à une distance bornée de son taux asymptotique peut être abstrait.

On définit $early_a$ comme le plus petit mot binaire infini (au sens de $\preceq$) respectant la propriété sur la position des 1 dans les mots abstraits par a. De même, $late_a$ est le plus grand mot binaire infini respectant la propriété sur la place des 0.

Définition 7 ($early_a$, $late_a$). *Soit $a = \left(b^0, b^1, r\right)$ une valeur abstraite.*

$$\begin{array}{lcl} early_a & = & \sqcap\{w,\ \forall i \geq 1,\ w[i] = 1 \Rightarrow \mathcal{O}_w(i-1) < r \times i + b^1\} \quad ✿ \\ late_a & = & \sqcup\{w,\ \forall i \geq 1,\ w[i] = 0 \Rightarrow \mathcal{O}_w(i-1) \geq r \times i + b^0\} \quad ✿ \end{array}$$

Tous les mots w appartenant à la concrétisation de a sont tels que $early_a \preceq w$ et $w \preceq late_a$. Par conséquent, si la concrétisation de a est non vide, alors $early_a \preceq late_a$. ✿

À l'inverse, si $early_a \preceq late_a$, alors $early_a$ et $late_a$ appartiennent à la concrétisation de a. ✿ Cela implique que la concrétisation est non vide, et que $early_a$ en est l'infimum et $late_a$ le supremum :

Proposition 1. ✿ $early_a \preceq late_a \Rightarrow (concr(a) \neq \varnothing) \wedge (early_a = \sqcap a) \wedge (late_a = \sqcup a)$

Par exemple, dans la figure 2, $w'_2 = \mathtt{00}(\mathtt{10110})$ est l'infimum de la concrétisation de a_2, les fronts montants sont effectués dès que possible, et $w''_2 = \mathtt{00}(\mathtt{01101})$ en est le supremum, l'absence de front est effectuée tant que possible (*i.e.* les fronts montants sont effectués aussi tard que possible). Il est montré dans la section 3.2, que ces deux horloges particulières peuvent être efficacement calculées par un programme synchrone.

Par conséquent, l'ensemble de concrétisation est vide si et seulement si le plus petit mot respectant la contrainte sur les 1 ne précède pas le plus grand mot vérifiant la contrainte sur les 0 (l'ensemble des mots vérifiant les conditions à la fois sur les 0 et les 1 est alors vide) : $concr(a) = \varnothing \Leftrightarrow early_a \not\preceq late_a$. ✿

Le nombre de 1 vus dans $early_a$ et $late_a$ sont définis par les formules suivantes : [3]

$$\begin{array}{lrcl} \forall i, & \mathcal{O}_{early_a}(i) & = & \max(0, \min(i, \lceil r \times i + b^1 \rceil)) \quad ✿ \\ & \mathcal{O}_{late_a}(i) & = & \max(0, \min(i, \lceil r \times i + b^0 \rceil)) \quad ✿ \end{array}$$

Celles-ci sont utilisées de manière déterminante dans les preuves.

La proposition suivante donne une condition suffisante pour assurer que l'ensemble de concrétisation d'une valeur abstraite a est non vide :

Proposition 2 (test de non vacuité)**.** ✿ $\forall a = \left(b^0, b^1, r\right),\ b^0 \leq b^1 \Rightarrow concr(a) \neq \varnothing$.

En effet, si la courbe est toujours strictement sous Δ^1 et/ou au dessus ou sur Δ^0, alors il existe toujours un élément valide à l'index i. C'est le cas quand Δ^1 est au dessus de ou égale à Δ^0, c'est à dire quand $b^1 \geq b^0$. Au contraire, dans la figure 3, la concrétisation de $a_3 = \left(\frac{2}{5}, 0, \frac{3}{5}\right)$ est vide : il n'y a pas d'élément valide au troisième instant car la courbe est au dessus de Δ^1 et en dessous de Δ^0.

La concrétisation contient un et un seul élément si $b^0 = b^1$. C'est le cas dans la figure 2 où w_1 est l'unique élément de la concrétisation de a_1. En effet, la courbe est toujours soit en dessous de Δ^1, soit sur ou au dessus de Δ^0 (mais pas les deux). Il n'y a alors à chaque indice qu'un seul élément valide.

Cette proposition n'est pas une équivalence. En effet, Δ^1 peut être légèrement au dessous de Δ^0 si la fonction $\mathcal{O}_w$ ne peut jamais prendre une valeur située entre les deux droites.

Nous avons un ordre partiel sur les horloges abstraites :

Définition 8 (relation d'ordre $\sqsubseteq^{\sim}$)**.** ✿ $ac_1 \sqsubseteq^{\sim} ac_2 \overset{def}{\Leftrightarrow} concr(ac_1) \subseteq concr(ac_2)$

Cette relation d'ordre peut être testée de manière efficace :

Proposition 3 (test $\sqsubseteq^{\sim}$)**.** ✿ *Soient* $a_1 = \left({b^0}_1, {b^1}_1, r_1\right)$ *et* $a_2 = \left({b^0}_2, {b^1}_2, r_2\right)$ *deux valeurs abstraites telles que* $concr(a_1) \neq \varnothing$ *et* $concr(a_2) \neq \varnothing$. *Alors*

$$(r_1 = r_2) \text{ et } ({b^0}_1 \geq {b^0}_2) \text{ et } ({b^1}_1 \leq {b^1}_2) \Rightarrow (a_1 \sqsubseteq^{\sim} a_2)$$

Nous noterons $abs(w)$ toute fonction telle que $abs(w) = a \Rightarrow w \in concr(a)$. [4]

3. $\lceil x \rceil$ est la partie entière supérieure de x.
4. Traditionnellement [13], les fonctions d'abstraction sont notées α et les fonctions de concrétisation γ.

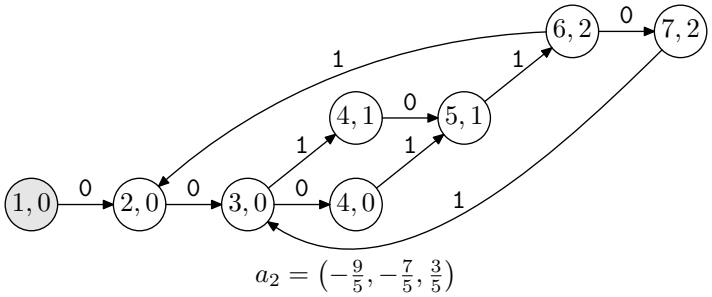


FIGURE 4 – Automates reconnaissant les mots représentés par a_2.

3.2. Abstractions et automates

Nous avons vu que la concrétisation d'une horloge abstraite est un ensemble de mots binaires qui peut être vide, contenir un seul mot ou une infinité de mots. Cet ensemble peut être représenté par un automate fini déterministe qui reconnaît tous les mots de l'ensemble et seulement eux. Nous commençons par définir un automate infini tel que le langage reconnu soit l'ensemble de concrétisation. Puis, nous montrons qu'il est équivalent à un automate fini.

Définition 9 (automate associé à une valeur abstraite)**.**
Soit une valeur abstraite $a = \left(b^0, b^1, r\right)$*. L'automate infini associé à* a *est* $I_a = \langle Q, \Sigma, \delta, q_o \rangle$ *avec :*
- *l'ensemble d'états* Q *est un ensemble de paires* $(i, j) \in \mathbb{N}^2$ *;*
- *l'état initial* q_o *est* $(1, 0)$ *;*
- *l'alphabet* Σ *est* $\{0, 1\}$ *;*
- *la fonction de transition* δ *est définie par :*
  $\delta(0, (i, j)) = (i + 1, j)$ *si* $j \geq r \times i + b^0$
  $\delta(1, (i, j)) = (i + 1, j + 1)$ *si* $j < r \times i + b^1$
  la fonction est non définie sinon.

Les étiquettes des états correspondent aux coordonnées dans les chronogrammes. La valeur i est l'indice de l'instant courant, et j est le nombre de $1s$ vus avant l'instant courant ($\mathcal{O}_w(i-1)$). Une transition de l'état (i, j) à l'état $(i + 1, j + 1)$ correspond à un front montant dans la courbe $\mathcal{O}_w$, *i.e.* l'occurrence d'un 1 à l'indice i. Cette transition peut être prise si le point (i, j) est sous la droite d'équation $r \times i + b^1$. Une transition de (i, j) à $(i+1, j)$ correspond à l'absence de front dans la courbe, *i.e.* l'occurrence d'un 0 à l'indice i. Elle est autorisée si le point (i, j) est au-dessus de ou sur la droite d'équation $r \times i + b^0$.

Pour se ramener à un automate fini, on peut remarquer que pour une valeur abstraite de taux $r = \frac{n}{l}$, tous les états de la forme $(i+x\times l, j+x\times n)$ avec $x \in \mathbb{N}$ sont équivalents. En effet, la condition $j+x\times n \geq r\times(i+x\times l)+b^0$ est équivalente à $j \geq r\times i+b^0$ et la condition $j+x\times n < r\times(i+x\times l)+b^1$ est équivalente à $j < r\times i+b^1$. Il y a donc un nombre fini d'états équivalents. La fonction de transition de l'automate fini déterministe A_a équivalent à l'automate infini I_a est :
$\delta(0, (i, j)) = nf(i + 1, j)$ si $j \geq r \times i + b^0$
$\delta(1, (i, j)) = nf(i + 1, j + 1)$ si $j < r \times i + b^1$
avec $r = \frac{n}{l}$, $nf(i, j) = (i - x \times l, j - x \times n)$, $x = \max\{x \in \mathbb{N}, (x \times l \leq i) \wedge (x \times n \leq j)\}$

Il est donc possible de représenter un ensemble de concrétisation infini par un automate fini. La figure 4 montre l'automate associé à $a_2 = \left(-\frac{9}{5}, -\frac{7}{5}, \frac{3}{5}\right)$ de la figure 2.

L'infimum de l'ensemble de concrétisation (selon $\preceq$) correspond au chemin dans l'automate où les transitions 1 sont prises prioritairement. Pour le supremum, ce sont les transitions 0 qui sont prises prioritairement. Pour une abstraction de taux $r = \frac{n}{l}$ tous les cycles sont de même longueur (l) et contiennent le même nombre de 1 (n). Choisir une transition 1 ne fait que retarder la transition 0 correspondante.

Ces automates sont intéressants car ils permettent de vérifier dynamiquement ou statiquement (avec du *model checking*) qu'une horloge est dans la concrétisation d'une valeur abstraite. Il est à remarquer qu'il n'est pas nécessaire de construire explicitement l'automate. Voici un exemple de programme LUCID SYNCHRONE [20] qui vérifie qu'une horloge `clk` est abstraite par la valeur (`b0`, `b1`, `r`).

```
type rat = { num: int; den: int; }
let norm { num = n; den = l; } i j = if i >= l && j >= n then (i - l, j - n) else (i, j)
let node check (b0, b1, r) clk = ok where
  rec i, j = (1,0) fby norm r (i+1) (if clk then j + 1 else j)
  and ok = if clk then (rat_of_int j) <: r *: (rat_of_int i) +: b1
           else (rat_of_int j) >=: r *: (rat_of_int i) +: b0
```

Les opérateurs suffixés par « : » (`+:`, `*:`, `<:`, etc.) sont les opérateurs sur les rationnels. La fonction `norm` calcule de façon incrémentale la forme normale de l'état `(i,j)` en utilisant le taux $\frac{n}{l}$. Le noeud `check` prend en entrée une abstraction et un flot de booléens `clk`. Il maintient la valeur courante de l'état. L'état est initialisé à la valeur `(1,0)`, puis à chaque instant, `i` est incrémenté, et `j` est incrémenté si l'horloge `clk` était vraie à l'instant précédent. La fonction de normalisation est appliquée à ce nouvel état. Puis, suivant la valeur courante de `clk`, on vérifie que l'on va faire une transition valide. En suivant le même principe, on peut également générer des horloges contenues dans une abstraction.

```
let node generate choice (b0, b1, r) = clk where
  rec i, j = (1,0) fby norm r (i+1) (if clk then j + 1 else j)
  and one = (rat_of_int j) <: (r *: (rat_of_int i) +: b1)
  and zero = (rat_of_int j) >=: (r *: (rat_of_int i) +: b0)
  and clk = choice one zero
```

Le noeud `generate` est paramétré par une fonction `choice` qui choisit quelle transition prendre lorsqu'une transition `1` et une transition `0` sont possibles. Ainsi, on peut par exemple définir les noeuds qui génèrent l'infimum et le supremum d'une valeur abstraite `a` :

```
let node early a = generate (fun x y -> x) a
let node late a = generate (fun x y -> not y) a
```

Il est aussi possible de définir une fonction de choix qui génère des horloges aléatoires dans `a`.

3.3. Opérateurs abstraits

Dans cette section, nous définissons les opérateurs sur les valeurs abstraites, correspondant aux opérateurs sur les mots définis en section 2. Calculer ces opérations dans le domaine abstrait se fait en temps et mémoire constants. Le résultat dans le domaine abstrait contient le résultat de l'opération dans le domaine concret.

Définition 10 (opérateur $\boldsymbol{on}^{\sim}$). ✿ *Définissons un opérateur* $\boldsymbol{on}$ *abstrait, noté* $\boldsymbol{on}^{\sim}$ *:*

$$\left(b^0{}_1, b^1{}_1, r_1\right) \ \boldsymbol{on}^{\sim} \ \left(b^0{}_2, b^1{}_2, r_2\right) = \left(b^0{}_{12}, b^1{}_{12}, r_{12}\right)$$

avec : $r_{12} = r_1 \times r_2, \quad b^0{}_{12} = b^0{}_1 \times r_2 + b^0{}_2, \quad b^1{}_{12} = b^1{}_1 \times r_2 + b^1{}_2 + \delta$
et $b^0{}_1 \leq 0$, $b^0{}_2 \leq 0$, $r_1 = \frac{n_1}{l_1}$, $b^1{}_1 = \frac{k_1^1}{l_1}$, $r_2 = \frac{n_2}{l_2}$, $b^1{}_2 = \frac{k_2^1}{l_2}$. $\delta = \frac{l_1 - 1}{l_1} \times \frac{n_2 - 1}{l_2}$.

L'opérateur abstrait $\boldsymbol{on}^{\sim}$ ne s'applique que sur des valeurs abstraites a dont la borne b^0 est négative ou nulle. On peut toujours se ramener à ce cas en considérant l'horloge abstraite la plus précise a' dont la borne b^0 est négative ou nulle et telle que $a \sqsubseteq^{\sim} a'$. Si $a = \left(b^0, b^1, r\right)$ alors $a' = \left(\min(0, b^0), b^1, r\right)$. ✿
Cet opérateur abstrait est correct :
$\forall w_1 \in concr(a_1),\ \forall w_2 \in concr(a_2),\ w_1 \ \boldsymbol{on} \ w_2 \in concr(a_1 \ \boldsymbol{on}^{\sim} \ a_2)$. ✿

Définition 11 (opérateur $\texttt{not}^{\sim}$). ✿ *Définissons un opérateur* $\texttt{not}$ *abstrait, noté* $\texttt{not}^{\sim}$ *:*

$$\texttt{not}^{\sim}\left(\left(b^0, b^1, r\right)\right) = \left(-b^1 - \varepsilon, -b^0 - \varepsilon, 1 - r\right)$$

avec $r = \frac{n}{l}, \quad b^0 = \frac{k^0}{l}, \quad b^1 = \frac{k^1}{l}, \quad \varepsilon = \frac{l-1}{l}$.

Cet opérateur abstrait est correct : $\forall w \in concr(a),\ \texttt{not}\ w \in concr(\texttt{not}^{\sim}\ a)$. ✿

Étant donnée une fonction d'abstraction des mots ($abs(w)$), on peut calculer l'abstraction des horloges composées de ces mots. Elle est définie récursivement ainsi :

Définition 12 (fonction d'abstraction des horloges). ✿

$$\begin{array}{lcl} abs(\texttt{not}\ w) & \overset{def}{\Leftrightarrow} & \texttt{not}^{\sim}\ abs(w) \\ abs(c_1\ \texttt{on}\ c_2) & \overset{def}{\Leftrightarrow} & abs(c_1)\ \texttt{on}^{\sim}\ abs(c_2) \end{array}$$

Cette abstraction des horloges est correcte : $c \in concr(abs(c))$. ✿

3.4. Relations abstraites

Nous définissons maintenant les relations sur les valeurs abstraites correspondant aux relations sur les mots définies section 2. Une relation est vérifiée dans le domaine abstrait seulement si elle est vérifiée pour tout couple de mots dans les concrétisations respectives des abstractions considérées.

Définition 13 (synchronisabilité abstraite). ✿ *Définissons une relation* $\bowtie$ *abstraite, notée* $\bowtie^{\sim}$ *:*

$$ac_1 \bowtie^{\sim} ac_2 \overset{def}{\Leftrightarrow} \forall w_1 \in concr(ac_1), w_2 \in concr(ac_2),\ w_1 \bowtie w_2$$

Cette relation peut être vérifiée sur les valeurs abstraites :

Proposition 4 (test de synchronisabilité).
Soient $a_1 = \left({b^0}_1, {b^1}_1, r_1\right)$ *et* $a_2 = \left({b^0}_2, {b^1}_2, r_2\right)$ *telles que* $concr(a_1) \neq \varnothing$ *et* $concr(a_2) \neq \varnothing$. *On a :*

$$a_1 \bowtie^{\sim} a_2 \Leftrightarrow r_1 = r_2$$

En effet, si deux horloges restent à une distance bornée de leur taux asymptotique, alors le nombre de 1 de la première horloge reste à une distance bornée du nombre de 1 de la seconde si et seulement si leur taux sont égaux.

L'abstraction contient toutes les informations nécessaires pour vérifier la synchronisabilité de deux horloges sur leur abstraction : $abs(c_1) \bowtie^{\sim} abs(c_2) \Leftrightarrow c_1 \bowtie c_2$.

Définition 14 (précédence abstraite). ✿ *Définissons une relation* $\preceq$ *abstraite, notée* $\preceq^{\sim}$ *:*

$$ac_1 \preceq^{\sim} ac_2 \overset{def}{\Leftrightarrow} \forall w_1 \in concr(ac_1), w_2 \in concr(ac_2),\ w_1 \preceq w_2$$

La relation de précédence sur des valeurs abstraites de concrétisation non vide est vérifiée si et seulement si l'horloge la plus tardive dans la première abstraction précède l'horloge la plus précoce dans la seconde :

$$\begin{array}{lcl} a_1 \preceq^{\sim} a_2 & \Leftrightarrow & \sqcup(concr(a_1)) \preceq \sqcap(concr(a_2)) \\ & \Leftrightarrow & late_{a_1} \preceq early_{a_2} \end{array}$$ ✿

Quand les valeurs abstraites sont synchronisables, la relation de précédence abstraite peut être vérifiée par une condition suffisante.

Proposition 5 (test de précédence). ✿ *Soient* $a_1 = \left({b^0}_1, {b^1}_1, r\right)$ *et* $a_2 = \left({b^0}_2, {b^1}_2, r\right)$. *On a :*

$${b^0}_1 \geq {b^1}_2 \Rightarrow a_1 \preceq^{\sim} a_2$$

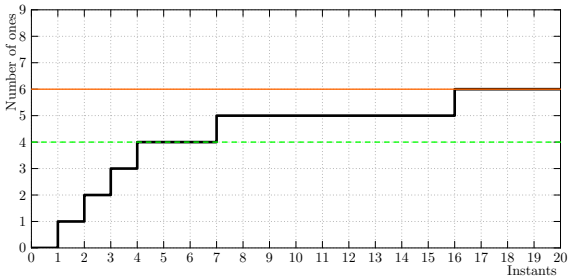


FIGURE 5 – La valeur abstraite $(4, 6, 0)$ représente des mots qui contiennent entre quatre et six 1 et une infinité de 0.

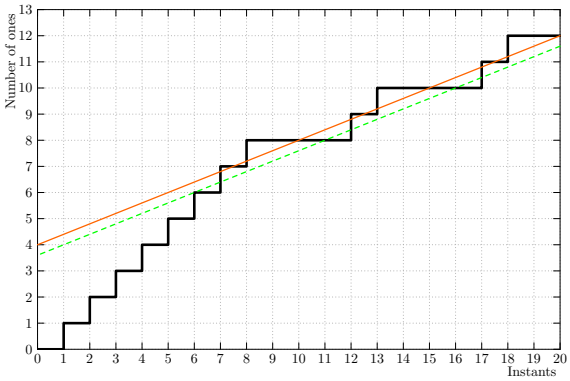


FIGURE 6 – Tous les mots représentés par la valeur abstraite $\left(\frac{18}{5}, 4, \frac{2}{5}\right)$ débutent par sept 1.

Pour vérifier la relation de précédence entre des horloges sur leur abstraction, le manque d'informations à propos du nombre de 1 (qui est seulement encadré) force à considérer le pire cas de concrétisation. Cette vérification sur les abstractions est donc correcte, mais pas complète par rapport à la vérification sur les horloges concrètes : $abs(c_1) \preceq^{\sim} abs(c_2) \Rightarrow c_1 \preceq c_2$. ✿

Nous définissons enfin la relation de sous-typage sur les valeurs abstraites :

Définition 15 (sous-typage abstrait)**.** ✿ *Définissons une relation* $<:$ *abstraite, notée* $<:^{\sim}$ *:*

$$ac_1 <:^{\sim} ac_2 \overset{def}{\Leftrightarrow} \forall w_1 \in concr(ac_1),\ w_2 \in concr(ac_2),\ w_1 <: w_2$$

Tout comme dans le domaine concret, cette relation est la conjonction des relations de synchronisabilité et de précédence : $a_1 <:^{\sim} a_2 \Leftrightarrow a_1 \bowtie^{\sim} a_2 \wedge a_1 \preceq^{\sim} a_2$. ✿

Il est donc aussi possible de tester efficacement la relation de sous-typage en utilisant les tests de synchronisabilité et de précédence.

Pour synchroniser les producteurs et les consommateurs, des buffers sont insérés. Une taille de buffer correcte pour communiquer de n'importe quelle horloge d'abstraction $a_1 = \left({b^0}_1, {b^1}_1, r\right)$ vers n'importe quelle horloge d'abstraction $a_2 = \left({b^0}_2, {b^1}_2, r\right)$, telles que $a_1 <:^{\sim} a_2$ est :

$$size(a_1, a_2) = \lceil {b^1}_1 - {b^0}_2 \rceil$$

4. Discussion

L'abstraction présentée ici apporte plusieurs avantages par rapport à l'abstraction à base d'*enveloppes* de [11]. En effet, l'abstraction à base d'enveloppes raisonne sur les indices des 1, et ne contraint que ceux-ci. Par conséquent, il est d'une part impossible de modéliser les mots ne contenant qu'un nombre fini de 1, et d'autre part impossible de forcer la présence d'une suite de 1 au début du préfixe de tous les mots représentés par la valeur abstraite. Ces deux inconvénients mènent à un opérateur abstrait $\boldsymbol{not}$ non défini quand le résultat concret contient un nombre fini de 1 (par exemple $\boldsymbol{not}\ 0000110111111110(1) = 1111001000000001(0)$), et qui perd de l'information ($a \sqsubseteq^{\sim} \boldsymbol{not}^{\sim}\ \boldsymbol{not}^{\sim}\ a$).

Ici, on raisonne sur le nombre de 1 vus dans le mot, et on permet ou non la présence d'un 1 et/ou d'un 0. D'une part, ceci donne la possibilité de représenter les horloges de taux asymptotique nul. Dans la figure 5, les horloges représentées par la valeur abstraite $(4, 6, 0)$ contiennent de quatre à six 1. Grâce à cela, l'opérateur $\boldsymbol{not}^{\sim}$ peut-être appliqué aux mots de taux 1 (dont la négation est un mot de

taux nul). D'autre part, cela permet de représenter des ensembles d'horloges qui débutent toutes par une suite de 1. Dans la figure 6, tous les mots représentés par la valeur abstraite $\left(\frac{18}{5}, 4, \frac{2}{5}\right)$ débutent par sept 1. Grâce à cela, l'application de l'opérateur $\mathit{not}^{\sim}$ ne perd aucune information sur la valeur abstraite (*i.e.* $\mathit{not}^{\sim}\ \mathit{not}^{\sim}\ a = a$).

Notons que la restriction ($b^0 \leq 0$) appliquée aux opérandes de l'opérateur $\mathit{on}^{\sim}$ n'est pas nouvelle, elle était intrinsèque à l'abstraction par enveloppes (où l'équivalent de cette condition est toujours vérifié). Se ramener à des opérandes traitées par $\mathit{on}^{\sim}$ revient à se ramener à une opération similaire au $\mathit{on}^{\sim}$ sur les enveloppes. Le résultat obtenu ici n'est donc pas plus précis que le résultat obtenu dans l'abstraction à base d'enveloppes. Nous pourrions imaginer la réalisation d'un opérateur $\mathit{on}^{\sim}$ plus précis, néanmoins c'est une tâche difficile car le préfixe du résultat comporte plusieurs phases. Par exemple, $1111011111(100)\ \mathit{on}\ 111(10) = 1111001010(100000)$. Une première phase du préfixe est une suite de 1, suit ensuite une phase où le préfixe du premier mot est échantillonné au taux $\frac{1}{2}$ et enfin vient le suffixe de taux $\frac{1}{6}$.

Enfin, on peut remarquer que toute enveloppe de [11] peut-être traduite en une valeur abstraite de notre abstraction, sans perte d'information : $[d, D]\,(T)$ se traduit en $\left(-\frac{D}{T}, -\frac{d}{T} - \varepsilon, \frac{1}{T}\right)$ avec $T = \frac{l}{n}$, $\varepsilon = \frac{n-1}{l}$. La traduction inverse est impossible si le taux est nul, et perd de l'information si $b^0 \geq 0$ (il faut d'abord se ramener à un $b^0 \leq 0$).

5. Applications

5.1. Abstraction d'horloges périodiques

Pour être capables de vérifier la relation de sous-typage et calculer la taille des buffers, l'usage exclusif de mots binaires ultimement périodiques à été proposée dans [10]. [5]

Le comportement périodique de ces mots permet de calculer statiquement les opérateurs on et not (les définitions deviennent des algorithmes). De même, il permet de vérifier les relations de précédence (si elle est vérifiée jusqu'à un certain rang, alors elle l'est pour toujours) et de synchronisabilité (qui est équivalente à l'égalité entre les taux de 1 du comportement périodique). Enfin, la définition de la taille de buffer nécessaire devient aussi un algorithme.

Cependant, il peut être intéressant d'éviter les calculs exacts sur les mots binaires infinis à cause de leur coût : par exemple, l'opérateur on nécessite un parcours complet des éléments des périodes données en argument. De plus, si les tailles des opérandes ne sont pas compatibles, le résultat est bien plus long que les arguments. Dans certains contextes comme les applications vidéos, ce coût pose problème car les tailles de périodes peuvent être énormes : dans l'exemple cité dans [10], un downscaler classique, l'horloge de sortie a un comportement périodique de longueur 17280. Si on ajoute un traitement spécifique à effectuer entre les images, on obtient un comportement périodique de longueur 2073600 (la taille d'une image haute définition). Calculer sur les valeurs abstraites donne une solution à ce problème.

Notons que quand des horloges périodiques sont utilisées, l'abstraction des mots peut-être automatiquement calculée :

Proposition 6 (fonction d'abstraction des mots binaires périodiques)**.**
Soit $w = u(v)$ *un mot binaire infini.* $abs(w) = \left(b^0, b^1, r\right)$ *avec :*
$r = \frac{|v|_1}{|v|}$ *avec* $|v|$ *la taille du mot binaire fini* v *et* $|v|_1$ *le nombre de* 1 *apparaissant dans* v*,*
$b^0 = \min_i(\mathcal{O}_w(i-1) - r \times i)$ *avec* $i \in \{x$ *tels que* $(1 \leq x \leq |u| + |v|)$ *et* $w[x] = 0\}$*,*
$b^1 = \max_i(\mathcal{O}_w(i-1) - r \times i) + \frac{1}{|v|}$ *avec* $i \in \{x$ *tels que* $(1 \leq x \leq |u| + |v|)$ *et* $w[x] = 1\}$*.*

5. Ces mots binaires infinis ont été utilisés depuis lors pour spécifier statiquement des ordonnancements périodiques pour le *Latency Insensitive Design* [7].

L'intuition derrière cette formule n'est autre que de parcourir le mot et d'élargir l'abstraction au fur et à mesure des besoins. Comme le mot est périodique, il suffit de parcourir le préfixe et une fois le suffixe pour obtenir une abstraction correcte.

Par exemple, l'abstraction de l'horloge de sortie du downscaler est :
$abs((\mathsf{10100100})\ \mathit{on}\ \mathsf{0}^{3600}(1)\ \mathit{on}\ (\mathsf{1}^{720}\mathsf{0}^{720}\mathsf{1}^{720}\mathsf{0}^{720}\mathsf{0}^{720}\mathsf{1}^{720}\mathsf{0}^{720}\mathsf{0}^{720}\mathsf{1}^{720}))$
$= \left(0, 0, \frac{3}{8}\right)\ \mathit{on}^{\sim}\ (-3600, -3600, 1)\ \mathit{on}^{\sim}\ \left(-400, \frac{4312}{9}, \frac{4}{9}\right) = \left(-2000, -1120, \frac{1}{6}\right)$

On peut enfin s'intéresser aux horloges affines, présentées dans [23, 22], qui sont un sous-ensemble des horloges périodiques, de la forme $\mathsf{0}^{\phi}(\mathsf{10}^{l-1})$. Elles ont été utilisées pour étendre le calcul d'horloges du langage synchrone flot de données Signal [5], dans le contexte du co-design logiciel/matériel. Grâce à la forme régulière de ces horloges, le calcul d'horloges étendu de Signal offre un algorithme d'unification plus puissant. Dans le cas des horloges affines, abstraire un mot est très simple : $abs(\mathsf{0}^{\phi}(\mathsf{10}^{l-1})) = \left(-\frac{\phi}{l}, -\frac{\phi}{l}, \frac{1}{l}\right)$ et ne perd aucune information. En effet, l'ensemble de concrétisation ne contient qu'un élément (comme spécifié en section 3, les bornes sont égales).

5.2. Spécifications à l'aide d'horloges abstraites

Nous avons vu dans les exemples de la section 3 que l'abstraction d'horloges permet de modéliser des horloges soumises à une gigue bornée. Elle permet de raisonner non plus sur une horloge précise mais sur une spécification de cette horloge : son abstraction. Toutes les horloges répondant à cette spécification ont un taux asymptotique commun et s'éloignent de ce taux dans la limite imposée par des bornes elles aussi communes. Cela permet de prouver des propriétés (comme l'absence de famine et de dépassement de capacité) sur le programme quelle que soit l'horloge effective, du moment qu'elle répond à sa spécification.

L'abstraction permet aussi de modéliser le temps d'exécution de processus temps-réel [14, 24, 16]. Pour spécifier qu'une fonction f est exécutée un cycle sur dix, et que son exécution prend entre deux et quatre cycles, on peut lui donner la signature $f : \forall\alpha.\ \alpha\ \mathtt{on}^{\sim}\ \left(0, 0, \frac{1}{10}\right) \to \alpha\ \mathtt{on}^{\sim}\ \left(-\frac{4}{10}, -\frac{2}{10}, \frac{1}{10}\right)$. Si on la compose avec elle même, on obtient $f \circ f : \forall\alpha.\ \alpha\ \mathtt{on}^{\sim}\ \left(0, 0, \frac{1}{10}\right) \to \alpha\ \mathtt{on}^{\sim}\ \left(-\frac{8}{10}, -\frac{4}{10}, \frac{1}{10}\right)$. L'exécution de f composée à elle même prend donc entre quatre et huit cycles.

Enfin, il est aussi possible de modéliser un trait courant des applications vidéo : la lecture ou l'écriture de plusieurs valeurs du même flot au sein d'un même instant logique [15]. Considérons, par exemple un noeud f produisant un flot de type d'horloges $\alpha\ \mathtt{on}^{\sim}\ (0, 0, 1)$ transmis à travers un buffer à un noeud g consommant un flot de type d'horloges $\alpha\ \mathtt{on}^{\sim}\ (-4, 0, 1)$. À chaque instant, f produit exactement une valeur. Quand à lui, g peut attendre quatre instants avant de commencer à consommer. Au cinquième instant, il peut alors lire les cinq premières valeurs, et continuer ensuite à consommer des « tableaux » de cinq valeurs un instant sur cinq. Cet exemple est illustré figure 7.

6. Conclusion

Cet article généralise la notion d'horloge dans les langages synchrones flot de données en permettant la manipulation d'ensembles d'horloges. Cette généralisation est basée sur l'introduction de valeurs abstraites qui permettent l'encadrement d'horloges. Nous nous sommes ici concentrés sur les propriétés algébriques de ces horloges et nous avons illustré leur expressivité. La motivation de ce travail est essentiellement pragmatique et donne une réponse à la nécessité de modéliser la gigue, le temps d'exécution et, plus généralement, la communication par buffers de taille bornée. La véritable nouveauté est de traiter des propriétés quantitatives lors du calcul d'horloges au lieu de se limiter à du synchronisme strict. Nous avons expérimenté l'usage de ces horloges sur plusieurs exemples (*picture-in-picture* vidéo et filtres de radio logicielle). L'extension du calcul d'horloges du compilateur de Lucid Synchrone [20] est actuellement en cours.

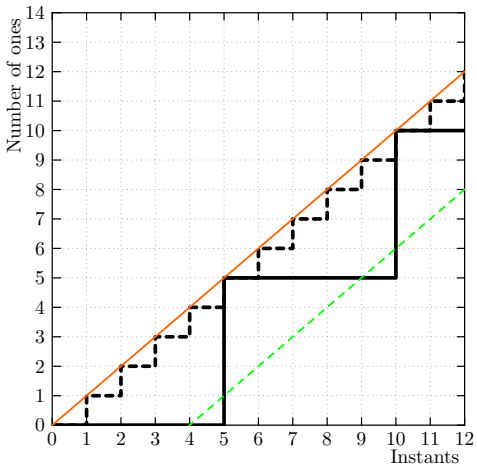


FIGURE 7 – Modélisation de la lecture/écriture de plusieurs valeurs au sein du même instant

Remerciements. Pascal Raymond nous a montré une façon inattendue (et très élégante) d'utiliser le compilateur LUSTRE pour générer des horloges dans une abstraction. Gwenaël Delaval nous a fourni un exemple de radio logicielle. Sylvie Boldo, Stéphane Lescuyer et Matthieu Sozeau ont toujours été très disponibles pour nous faire découvrir COQ. Les figures ont été programmées en Mlpost, nous remercions Johannes Kanig et Stéphane Lescuyer pour leur aide. Nous remercions Marc Pouzet pour ses remarques constructives sur l'abstraction. Enfin, nous remercions les rapporteurs anonymes et le comité de programme.

Références

[1] T. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Programming Languages Design and Implementation*, pages 163–173. ACM, 1995.

[2] André Arnold. *Systèmes de transitions et sémantique des processus communicants*. Masson, 1992.

[3] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[4] Albert Benveniste and Gérard Berry. *The synchronous approach to reactive and real-time systems*, pages 147–159. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[5] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations : the SIGNAL language and its semantics. *Sci. Comput. Program.*, 16(2) :103–149, 1991.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development Coq'Art : The Calculus of Inductive Constructions*. Springer-Verlag. Series : Texts in Theoretical Computer Science. An EATCS Series, 2004.

[7] Julien Boucaron, Robert de Simone, and Jean-Vivien Millo. Formal methods for scheduling of latency-insensitive designs. *EURASIP Journal on Embedded Systems*, Issue 1(ISSN :1687-3955) :8 – 8, January 2007.

[8] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy : A framework for simulating and prototyping heterogeneous systems. *International Journal of computer Simulation*, 1994. special issue on Simulation Software Development.

[9] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating Discrete-Time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 2005. Special Issue on Embedded Software.

[10] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-Synchronous Kahn Networks : a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages*, January 2006.

[11] Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. In *The Sixth ASIAN Symposium on Programming Languages and Systems (APLAS 08)*, Bangalore, India, December 2008.

[12] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software*, Philadelphia, USA, October 2003.

[13] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[14] Adrian Curic. *Implementing Lustre Programs on Distributed Platforms with Real-time Constraints*. PhD thesis, Université Joseph Fourier, 2005.

[15] Léonard Gérard. Des horloges entières pour la répartition de programmes synchrones flot de données. Rapport de master, Université Paris-Sud 11, 2008.

[16] Nicolas Halbwachs and Louis Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Sixth International Conference on Application of Concurrency to System Design*, Turku, Finland, June 2006.

[17] E. Lee and D. Messerschmitt. Synchronous dataflow. *IEEE Trans. Comput.*, 75(9), 1987.

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, 1978.

[19] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3) :267–310, 1983.

[20] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at : `www.lri.fr/~pouzet/lucid-synchrone`.

[21] Scicos. `http://www-rocq.inria.fr/scicos`.

[22] Irina M. Smarandache, Thierry Gautier, and Paul Le Guernic. Validation of mixed SIGNAL-ALPHA real-time systems through affine calculus on clock synchronisation constraints. In *World Congress on Formal Methods (2)*, pages 1364–1383, 1999.

[23] Irina M. Smarandache and Paul Le Guernic. Affine transformations in SIGNAL and their application in the specification and validation of real-time systems. In *ARTS*, 1997.

[24] Christos Sofronis. *Embedded Code Generation from High-level Heterogeneous Components*. PhD thesis, Université Joseph Fourier, 2006.

[25] Jean Vuillemin. On Circuits and Numbers. Technical report, Digital, Paris Research Laboratory, 1993.

Faut-il avoir peur de sa carte SIM ?

Bruno Barras

Trusted Labs

Les cartes à puces sont partout : dans les téléphones mobiles, les cartes bancaires ou encore dans les titres de transport. D'abord complètement fermées, la nouvelle génération de cartes à puces ont considérablement évolué technologiquement : chargement dynamique d'applications, présence d'entités potentiellement concurrentes sur une même carte ... ce qui n'est pas sans éveiller des craintes concernant la sécurité de ces systèmes complexes.

Dans cet exposé, nous présenterons brièvement les problèmes rencontrés lorsque l'on souhaite modéliser formellement (à l'aide de l'assistant à la preuve Coq) une carte à puce, et prouver des propriétés de sécurité comme l'intégrité et la confidentialité des données sensibles d'une application.

Nous aborderons plus en détail la description d'une méthodologie basée sur l'utilisation de types dépendants, permettant d'une part de spécifier de manière simple et uniforme les transitions de ces systèmes, et d'autre part de faciliter les preuves fastidieuses de séparation des différentes composantes de l'état du système.

Fouille au code OCaml par analyse de dépendances

Maxence Guesdon[1]

1: Service Expérimentations et Développements
INRIA Paris-Rocquencourt
Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France
maxence.guesdon@inria.fr

Résumé

Dans le cadre de la reprise d'un logiciel existant écrit en OCaml, une remise au propre de ce code s'avère nécessaire avant de poursuivre le développement du logiciel. Dans ce but, un outil permettant de trouver les éléments utilisés ou non serait utile pour gagner du temps.

Nous présentons une analyse de code OCaml basée sur la construction d'un graphe de dépendances annoté. Les sommets de ce graphe sont les éléments du langage OCaml (valeurs, modules, types, ...) et les annotations sur les arcs permettent de préciser les dépendances (utilisation, héritage, ...).

Par la suite, un langage de sélection des éléments de ce graphe est défini, permettant de trouver les éléments non référencés (révélateurs de code potentiellement inutile), d'autres éléments comme les champs de types enregistrements jamais consultés.

Cette analyse est implémentée et disponible dans le logiciel libre Oug. Quelques traitements statistiques sur le graphe annoté sont évoqués pour tenter de faciliter la compréhension de l'organisation du code analysé.

1. Introduction

Le travail présenté se situe dans le cadre de la reprise d'un logiciel existant, que nous appellerons L, écrit en OCaml. Une remise en forme du code de L et la fusion de deux branches contenant des évolutions majeures étaient nécessaires.

Pour cette tâche, un outil d'analyse de dépendances dans du code OCaml permettrait un gain de temps appréciable. Nous présentons dans la section 2 les détails du problème, des analyses existantes s'en rapprochant ainsi que la solution envisagée, basée sur la construction d'un graphe de dépendances entre les différents éléments du langage (valeurs, modules, classes, etc.).

La section 3 présente la construction de ce graphe, enrichie au fur et à mesure des tests et des réflexions. En section 4, nous expliquons comment exploiter ce graphe pour effectuer les analyses qui nous intéressent, notamment en définissant un langage de sélection de sommets (éléments) sur le graphe. Enfin, la section 5 donne des informations sur l'implémentation de cette analyse dans un outil en ligne de commande et le résultat de l'utilisation de ce dernier sur le logiciel L.

2. Le besoin

2.1. Le problème

Notre logiciel L contient presque 60.000 lignes de code OCaml écrites par plusieurs personnes successives avec des styles différents (cas classique de CDD à répétition).

Le résultat est :

1. un code peu homogène,
2. des redondances (le même calcul implémenté dans plusieurs fonctions, par exemple), avec comme cause principale la méconnaissance et le manque d'organisation claire de l'existant,
3. du code inutile, qui provoque tout de même des erreurs de compilation lors de changements dans les structures de données et est donc maintenu pour passer l'étape de la compilation,
4. des champs d'enregistrements qui ne sont plus utiles et qui pourtant sont calculés,
5. des constructeurs qui ne sont plus utiles mais sont encore présents comme autant de cas de pattern-matching.

Le code inutile évoqué au point 3 peut être du code mort, c'est-à-dire jamais exécuté, comme la fonction `f` dans l'exemple suivant :

```
let f () = print_string "hello";;
(* fin du programme *)
```

Le code inutile inclut également du code calculant une valeur qui n'est jamais utilisée par la suite, comme la valeur `foo` dans :

```
let f x = x + 1;;
let foo = f 1;;
(* la suite du programme n'utilise pas foo *)
```

Nous souhaitons donc pouvoir trouver les éléments non référencés (donc inutiles), révélateurs de code *potentiellement* inutile. En effet, le code des éléments non référencés est inutile si ce code est purement fonctionnel, mais seulement *potentiellement* inutile s'il contient des effets de bord. Ainsi, dans le code suivant

```
let f x = print_int x ; x + 1;;
let g x = let foo = f x in x + 1;;
```

la valeur `foo` est inutile mais, son calcul ayant un effet de bord, on ne peut simplement supprimer "`let foo = f x in`" sans changer la sémantique du programme.

Les points 1 à 5 représentent un coût supplémentaire en terme de temps de développement et de maintenance. De plus, les points 3, 4 et 5 ont un impact sur les performances.

Une remise au propre du code de L implique d'intervenir sur ces différents points. L'amélioration de l'homogénéité du code et la détection de redondances nécessitent une bonne connaissance de L ; cependant, cette connaissance est plus facile à acquérir sans les entraves que représentent les autres points, que l'on souhaite donc traiter en priorité.

La recherche du code et des champs inutiles étant fastidieuse à la main, on souhaite avoir un outil pour l'automatiser. De plus, un tel outil permettrait de contrôler régulièrement l'apparition de telles anomalies pour les traiter rapidement.

Un autre problème de L est l'existence de deux développements majeurs en parallèle, qu'il faut fusionner. Malheureusement, l'un de ces développements, D_1, introduit de nouvelles structures

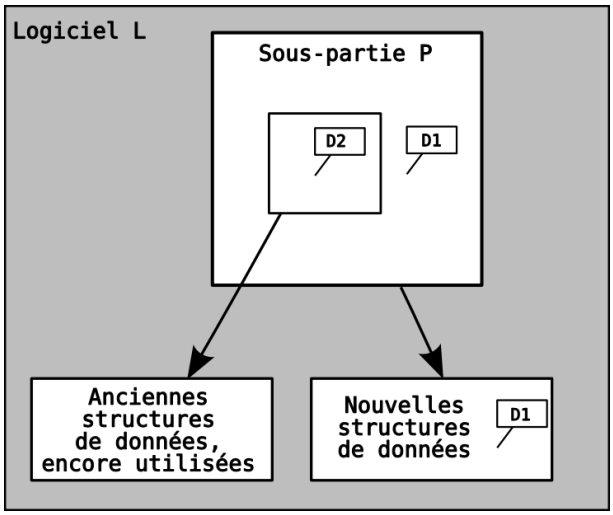


FIGURE 1 – Situation du code juste après la fusion des deux développements réalisés en parallèle. On souhaite que le code du développement D_2 utilise maintenant les nouvelles structures de données comme le reste de la partie P du logiciel.

de données utilisées dans une sous-partie P du logiciel à la place d'anciennes, tandis que l'autre développement, D_2, impacte la même sous-partie P. Les anciennes structures de données sont encore utilisées et la plupart des données sont dans une référence globale à l'application, ce qui ne permet pas de s'appuyer sur le typage pour détecter, dans le deuxième développement, le code utilisant les anciennes structures alors qu'il devrait utiliser les nouvelles. La situation est représentée sur la figure 1. On souhaite donc savoir où les modules concernés par D_2 accèdent aux anciennes structures de données, soit directement, soit, plus souvent, par l'intermédiaire de fonctions qui, elles, accèdent à ces anciennes structures. Cette problématique se rapproche de la recherche des champs inutilisés, mais inversée : il s'agit de savoir, pour chaque champ des anciennes structures, quels éléments de D_2 y font référence directement ou indirectement.

2.2. Les analyses existantes

Notre but est de construire une représentation du logiciel L pour en extraire les informations dont nous avons besoin pour faciliter la compréhension du code et supprimer les parties inutiles.

Les techniques de suppression de code mort basées sur des analyses locales de vivacité et la propagation de constantes comme dans [1] ne nous conviennent donc pas, d'autant plus qu'elles sont souvent faites sur du code généré et non sur du code source.

Une analyse de dépendances sur du code ML est exposée dans [3] mais vise à ordonner les modules pour décharger le développeur de cette tâche. On y retrouve les problèmes de portée et de masquage que nous évoquons plus loin.

La création d'un graphe de dépendances (pour abstraire la structure d'un programme C) et la possibilité de requêtes sur ce graphe (pour faciliter la maintenance d'un logiciel) sont décrites dans [2].

La solution que nous avons adoptée est proche de cette dernière méthode. Cependant, l'aspect premier ordre du langage OCaml rend difficile la création d'un graphe d'appels, qui nécessiterait une analyse du flot de données. D'autre part, les constructions d'OCaml nécessitent de définir des dépendances entre éléments qui soient adaptées au langage et à nos besoins.

2.3. L'idée de solution envisagée

Il n'existe aucun outil pour effectuer les analyses et recherches dont nous avons besoin pour du code OCaml. Une façon de faire est d'utiliser la commande UNIX `grep` mais le résultat affiché nécessite d'aller voir le code en question pour étudier dans quel contexte un identifiant recherché est utilisé.

Les fichiers `.annot` produits par l'option `-annot` du compilateur OCaml pourraient représenter une autre piste pour obtenir les informations nécessaires mais ils ne contiennent pas (encore) l'information indiquant où est déclarée un identifiant, et de toutes façons ils ne donnent pas le contexte d'utilisation, par exemple si un champ d'enregistrement est lu ou modifié.

A noter que le compilateur OCaml contient déjà la détection des variables non utilisées, mais il ne s'agit que des variables "locales", c'est-à-dire celles dans une classe ou une autre variable. Ainsi, dans le code suivant :

```
let foo =
  let (y, z) = (3, 4) in
  z;;
let bar gee = 3;;
```

les variables `y` et `gee` sont indiquées comme non utilisées, mais pas `foo` ni `bar`. Cela s'explique facilement puisque ces deux variables sont encore dans la portée lexicale du code qui suit et même accessibles depuis un autre module. En présence d'un fichier `.mli`, le compilateur pourrait détecter que ces variables ne sont ni utilisées dans le module ni présentées dans l'interface, mais cette fonctionnalité n'est pas encore disponible.

L'idée, assez évidente, pour détecter les cas présentés en section 2.1 est donc de construire un graphe de dépendances entre les différents éléments du code du logiciel L. Les types d'éléments qui nous intéressent sont les suivants : modules, valeurs, classes, méthodes, variables d'instance, types, champs de type (champ pour un type enregistrement, constructeur pour un type variant). Nous laissons de côté aujourd'hui les exceptions et les types variants polymorphes.

A partir de ce graphe, nous pourrons ensuite rechercher les éléments qui ne sont pas référencés afin de répondre aux points 3, 4 et 5 évoqués plus haut, ainsi que les éléments utilisant des champs de types donnés, dans le cadre de la problématique de fusion de code exposée précédemment.

Présentons maintenant la construction de ce graphe.

3. Construction du graphe

L'analyse de dépendances dont nous avons besoin nécessite d'analyser le code d'implémentation (les fichiers `.ml`) de notre logiciel L. Nous verrons dans la suite que nous ajouterons un traitement supplémentaire utile pour les bibliothèques, utilisant les fichiers d'interface `.mli`.

Les sommets du graphe représenteront les éléments (modules, valeurs, etc.) tandis que les arcs représenteront les relations de dépendances entre les éléments.

Dans un premier temps, on considère seulement les éléments "exportables", c'est-à-dire nommables dans le fichier `.mli` correspondant au fichier `.ml` analysé. Le code suivant donne quelques exemples d'éléments exportables ou non :

```
let (x, y) = let z = 1 in (z+1,0);; (* x et y sont exportables, z ne l'est pas *)
module P = struct (* P est exportable ... *)
  type t = int  (* ... de même que t ... *)
  let compare = Pervasives.compare (* ... et que compare *)
end;;
```

Code	Relations
`let x = 3 ; ; let y = x + 1 ; ;`	$y \rightarrow x$
`let y = x + 42 ; ;`	$y_2 \rightarrow x$
`let f x = x + y ; ;`	$f \rightarrow y_2$
`module M = struct`	
`  let g x = f (f (x + 2))`	$M \rightarrow M.g$, $M.g \rightarrow f$
`  let h t u = f (x + t) + g u`	$M \rightarrow M.h$, $M.h \rightarrow x$, $M.h \rightarrow f$, $M.h \rightarrow M.g$
`end ; ;`	
`type t = One | Two ; ;`	$t \rightarrow t.One$, $t \rightarrow t.Two$
`let rec int_of_t = function`	
`  One -> 1`	$int_of_t \rightarrow t.One$
`  | Two -> 1 + int_of_t One ; ;`	$int_of_t \rightarrow t.Two$, $int_of_t \rightarrow int_of_t$

FIGURE 2 – Production d'arcs de dépendance simples.

```
let f x = (* f est exportable *)
  let module M = Set.create(P) in (* M ne l'est pas *)
  ...;;
module S = Set.Make (* S est exportable *)
  (struct type t = int let compare = Pervasives.compare end)
  (* ce module anonyme n'est pas exportable *)
;;
```

3.1. Graphe simple

Une première approche, naïve, consiste à utiliser des arcs simples pour indiquer que, d'une façon ou d'autre autre, un élément `t` dépend de `u` (relation notée `t` $\rightarrow$ `u`). La figure 2 donne quelques exemples de code et des relations qui en sont extraites. De plus, si le code de cet exemple est dans un fichier `f.ml`, il faut également ajouter le module résultant `F` et les relations de dépendances vis-à-vis des éléments qu'il contient, en préfixant le nom de ces derniers. On obtient au final : `F` $\rightarrow$ `F.x`, `F` $\rightarrow$ `F.y`, `F.y` $\rightarrow$ `F.x`, `F` $\rightarrow$ `F.M`, `F.M` $\rightarrow$ `F.M.g`, etc. On peut remarquer que, même si `F.M.g` contient plusieurs références à `F.f`, un seul arc est ajouté. En effet, il est inutile de garder les références multiples entre deux éléments dans notre cas. D'autre part, la définition d'une seconde valeur `F.y` crée un sommet supplémentaire dans le graphe, distinct du sommet correspondant à la première valeur `F.y`, que nous appelons `F.`y_2 pour le distinguer à l'affichage dans l'exemple. Nous verrons comment ce phénomène de masquage est traité dans l'implémentation.

Ces arcs "simples" ne permettent cependant pas de mener l'analyse que nous souhaitons. En effet, puisque chaque élément (à part les modules de plus haut niveau) est contenu dans un autre élément, chaque élément a donc un prédécesseur. On pourrait imaginer ne pas ajouter les arcs représentant la relation de contenance d'un élément par un autre, mais on ne pourrait alors dire qu'un module est utile car il contient un élément utilisé.

Il nous faut donc préciser les types de dépendances entre éléments.

3.2. Graphe annoté

Nous définissons plusieurs types d'annotations pour les arcs, afin de garder des informations plus précises sur les relations entre éléments :

– `t` $\xrightarrow{contain}$ `u` indique que la définition de l'élément `t` contient la définition de l'élément `u` ; c'est le cas dans l'exemple précédent pour `F` et `F.x`, la définition de `F` contient la définition de `F.x` :

Code	Relations
`let x = 3 ; ; let y = x + 1 ; ;`	F.y $\xrightarrow{use}$ F.x
`let y = x + 42 ; ;`	F.y_2 $\xrightarrow{use}$ F.x
`let f x = x + y ; ;`	F.f $\xrightarrow{use}$ F.y_2
`module M = struct`	
`  let g x = f (f (x + 2))`	F.M $\xrightarrow{contain}$ F.M.g, F.M.g $\xrightarrow{use}$ F.f
`  let h t u = f (x + t) + g u`	F.M $\xrightarrow{contain}$ F.M.h,
	F.M.h $\xrightarrow{use}$ F.x, F.M.h $\xrightarrow{use}$ F.f, F.M.h $\xrightarrow{use}$ F.M.g
`end ; ;`	
`type t = One | Two ; ;`	F.t $\xrightarrow{contain}$ F.t.One, F.t $\xrightarrow{contain}$ F.t.Two
`let rec int_of_t = function`	
`  One -> 1`	F.int_of_t $\xrightarrow{use}$ F.t.One
`  | Two -> 1 + int_of_t One ; ;`	F.int_of_t $\xrightarrow{use}$ F.t.Two,
	F.int_of_t $\xrightarrow{use}$ F.int_of_t
`module A =`	
`  struct let x = 1 let y = x end ; ;`	F.A $\xrightarrow{contain}$ F.A.x, F.A $\xrightarrow{contain}$ F.A.y
`module B = A ; ;`	F.B $\xrightarrow{alias}$ F.A
`let g x = x + B.y ; ;`	F.g $\xrightarrow{use}$ F.A.y (car `F.B` est un alias pour `F.A`)
`module Fo`	
`  (P : sig val p : int -> int end) =`	
`  struct let f x = P.p x + A.x end ; ;`	F.Fo $\xrightarrow{contain}$ F.Fo.f, F.Fo.f $\xrightarrow{use}$ F.A.x
`module C = struct let p x = x + 1 end ; ;`	F.C $\xrightarrow{contain}$ F.C.p
`module D = Fo(C) ; ;`	F.D $\xrightarrow{include}$ F.C, F.D $\xrightarrow{include}$ F.Fo, F.D $\xrightarrow{contain}$ F.D.f,
	F.D.f $\xrightarrow{alias}$ F.Fo.f, F.D.f $\xrightarrow{use}$ F.C.p, F.D.f $\xrightarrow{use}$ F.A.x
`module E = struct include D end ; ;`	F.E $\xrightarrow{include}$ F.D
`let h = E.f ; ;`	F.h $\xrightarrow{use}$ F.D.f (car `F.E.f` est en fait `F.D.f`)

FIGURE 3 – Production d'arcs de dépendance annotés.

F $\xrightarrow{contain}$ F.x.

- `t` $\xrightarrow{include}$ `u` indique que l'élément `t` inclut l'élément `u` ; c'est le cas pour les inclusions de modules, ainsi que pour l'application de foncteurs : on considèrera que le foncteur et les arguments sont dans le module résultant (il n'est pas apparu le besoin d'avoir une relation spéciale pour lier un module `M` et les arguments passés au foncteur pour créer `M`).
- `c1` $\xrightarrow{inherit}$ `c2` indique une relation entre deux classes : `c1` hérite de `c2`.
- `t` $\xrightarrow{alias}$ `u` indique que l'élément `t` est une sorte d'alias pour l'élément `u`. Cette relation apparaît lors de l'application d'un foncteur. Le module résultat contient des éléments qui sont des aliases des éléments du foncteur appliqué. Par ailleurs, le cas où un module est créé par "`module B = A`" entraîne la création d'un arc "alias" pour conserver cette information : `B` $\xrightarrow{alias}$ `A`.
- `t` $\xrightarrow{use}$ `u` indique que l'élément `t` dépend de `u` d'une autre manière que celles ci-dessus, notamment quand une valeur en utilise une autre dans sa définition.

La figure 3 montre ce que devient l'exemple utilisé pour illustrer la construction du graphe sans annotation, complété par des exemples de manipulations de modules, en supposant que le code se trouve dans un fichier `f.ml`.

On peut remarquer que même si l'application d'un foncteur d'une part et la création d'un module par `include` explicite (`"module E = struct include D end"`) d'autre part entraînent toutes les deux la création d'arcs $\xrightarrow{include}$, un traitement différent est fait pour les éléments des modules inclus. Dans le cas d'un `include` explicite, le nouveau module ne contient aucun élément. En effet, ajouter des éléments au module `E` n'est pas utile puisque ces éléments auront les mêmes dépendances que les éléments du module `D` inclus.

Au contraire, dans le cas d'un module créé par application d'un foncteur, on crée pour ce module résultant autant d'éléments qu'en définit le foncteur, et chacun de ces éléments créés est lié à l'élément original du foncteur par un arc $\xrightarrow{alias}$. Cela est nécessaire car les éléments du module résultant de l'application n'auront pas forcément les mêmes dépendances que les éléments d'un autre module résultant de l'application du même foncteur mais avec un module différent en paramètre. La relation d'alias permettra de voir si un élément d'un foncteur est utilisé, car tant que le foncteur n'est pas appliqué, ses éléments ne peuvent être référencés depuis l'extérieur du foncteur.

De la même façon, la création d'un module `B` par `"module B = A"` n'entraîne pas la création d'éléments dans `B` car il ne s'agit que de la création d'un raccourci de nommage. Il serait toutefois possible d'appliquer le même traitement que pour l'application de foncteur, comme si le module `A` était un foncteur sans argument.

Ces annotations nous permettent donc de trouver :
- les éléments exportables qui ne sont pas utilisés (c'est-à-dire les éléments qui n'ont aucun prédécesseur par certaines relations, comme nous le détaillons en section 4.2),
- les éléments qui, directement ou transitivement, utilisent certains champs de types.

Remarque On ne prend pas en compte dans le graphe les paramètres des fonctions, c'est-à-dire que ces derniers n'ont pas de sommet dans le graphe pour les représenter. L'aspect ordre supérieur d'OCaml n'est donc pas directement abordé, comme le montre l'exemple suivant :

```
let conv = int_of_string;;
let f g x = print_int (g x);;
let h s = f conv s;;
```

Le graphe montrera les dépendances `h` $\xrightarrow{use}$ `conv` et `h` $\xrightarrow{use}$ `f` mais pas `f` $\xrightarrow{use}$ `conv`. Cet aspect est évoqué dans la conclusion.

3.3. Trouver les champs de type inutiles

Les annotations définies plus haut ne résolvent pas complètement les points 4 et 5 exposés en section 2.1 concernant l'utilité des champs de types définis dans le code existant.

En effet, OCaml ne permet pas la création d'une valeur de type enregistrement si tous les champs du type ne sont pas définis. Si au moins une valeur de ce type est créée, il existe alors toujours au moins une valeur qui définit tous les champs. En conséquence, il n'est pas possible avec les annotations d'arcs définies plus haut de voir si un champ d'un enregistrement est utilisé et pas seulement initialisé ou modifié.

Par ailleurs, le compilateur OCaml permet de savoir quand un pattern-matching est incomplet. Une bonne pratique de codage consiste à indiquer explicitement tous les constructeurs d'un type variant, même si plusieurs se traitent de la même façon, afin de s'assurer du traitement de tous les cas lors de l'ajout d'un constructeur. Ainsi, de façon inverse aux champs d'un type enregistrement, les constructeurs d'un type variant sont souvent tous filtrés à un moment ou à un autre. Le problème vient cette fois du fait que l'on recherche les constructeurs qui ne sont pas utilisés pour créer une valeur du type en question.

	Champ d'un enregistrement	Constructeur d'un type variant
"Lecture"	Lecture d'un champ : **v.field** `+ 1`	Filtrage : match x with **Const** → ...
"Création"	Initialisation/affectation : **v.field** ← `1`	Construction : `let x =` **Const** `in ...`

FIGURE 4 – Cas de "lecture" et "création" de champs de types.

Il nous faut donc distinguer la dépendance par rapport à un champ de type selon qu'on le "crée" ou bien qu'on le "lit". La figure 4 précise le sens de "créer" et "lire" selon le type de champ concerné. La "lecture" d'un champ entraîne la création d'un arc $\xrightarrow{use}$, tandis que la "création" d'un champ provoque l'ajout d'un arc avec une nouvelle annotation, $\xrightarrow{create}$. Pour trouver les champs d'enregistrement inutiles, on cherchera donc ceux qui n'ont aucun prédécesseur par la relation $\xrightarrow{use}$. Le cas où aucune valeur d'un champ enregistrement n'est créée sera détecté en cherchant les champs sans prédécesseur par la relation $\xrightarrow{create}$.

Pour les constructeurs, sous l'hypothèse que le développeur s'est attaché à traiter explicitement chaque constructeur lors des filtrages, on s'intéressera aux constructeurs qui ne sont jamais utilisés pour créer une valeur du type en question. Ce sont les champs de type variant qui n'ont aucun prédécesseur par la relation $\xrightarrow{create}$. Cependant, cette détection n'est pas tout à fait complète, comme le montre l'exemple suivant :

```
type t = One | Two ;;
let is_two x = x = Two;;
```

Ici, le constructeur `Two` est en position de "création" et non en position de filtrage. Si on teste le cas de `Two` à l'aide de la fonction `is_two` et que ce constructeur n'est utilisé nulle par ailleurs pour créer une valeur du type `t`, le graphe ne permettra pas de s'en rendre compte.

La figure 5 montre le graphe de dépendances annoté pour l'exemple de la figure 3, avec les arcs $\xrightarrow{create}$.

Le graphe avec ces annotations nous permet donc de détecter les éléments exportables qui nous intéressent (modulo l'exception ci-dessus). Cependant, il ne permet pas de détecter d'autres éléments pourtant inutiles, parmi les éléments non "exportables".

3.4. Gestion des éléments non exportables

Considérons le code suivant :

```
let x =
  let module M = struct let y = 1 end in
  let y = Array.create 12 0 in
  1;;
```

Nous avons clairement deux variables inutilisées et inutilisables dans la suite du code : `M.y` et `y`. Le compilateur OCaml ne signale pas le fait que `M.y` est inutilisée. Par ailleurs, le module `M` n'étant pas exportable puisque local à la variable `x`, il n'apparaît pas dans notre graphe. Il faut pourtant détecter ce cas. De même, notre graphe ne prend pas en compte la variable `y`, locale à `x` et non exportable, même si le compilateur OCaml émet un avertissement pour signaler qu'elle est inutilisée.

Ce genre de cas peut être facilement détecté en ajoutant dans notre graphe les sommets correspondant à ces éléments locaux. Pour le dernier exemple ci-dessus, on aura alors les relations suivantes, en utilisant des arcs $\xrightarrow{contain}$ pour indiquer qu'un élément est défini dans un autre élément de façon générale et plus seulement pour les modules, classes et types :

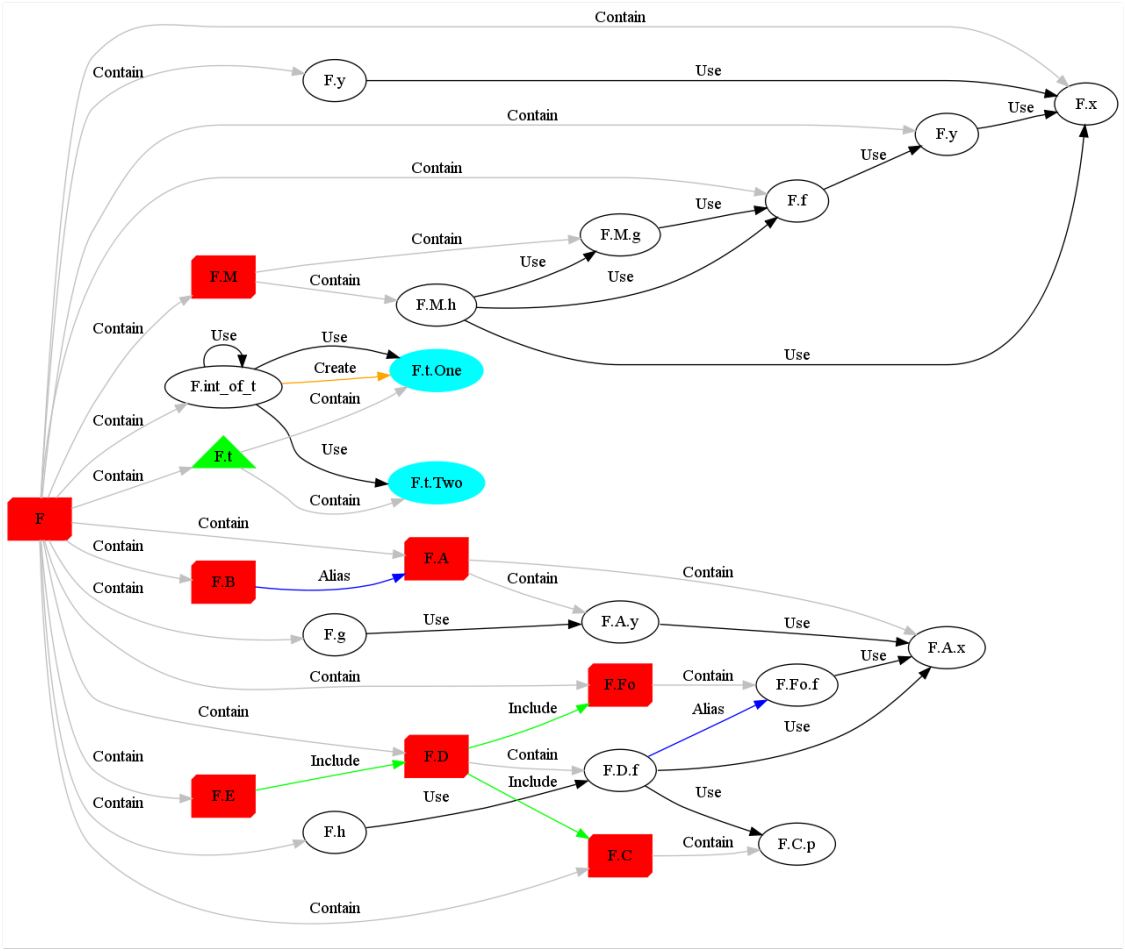


FIGURE 5 – Graphe annoté des éléments exportables de l'exemple.

$\texttt{x} \xrightarrow{contain} \texttt{x.M}$, $\texttt{x.M} \xrightarrow{contain} \texttt{x.M.y}$, $\texttt{x} \xrightarrow{contain} \texttt{x.y}$.

Il sera alors facile de détecter que `x.M.y` et `x.y` sont inutilisées, de même que le module `x.M` qui ne contient aucun élément utilisé (ce que n'indique pas le compilateur OCaml).

3.5. Les interfaces entrent en scène

L'analyse du code d'implémentation permet de trouver les éléments inutilisés. Cependant, dans le cas d'une bibliothèque, beaucoup d'éléments sont souvent inutilisés dans la bibliothèque elle-même mais accessibles dans son interface.

Pour éviter de mélanger les éléments non utilisés car seulement offerts par l'interface et ceux correspondant effectivement à du code inutile, il nous faut prendre en compte les fichiers d'interface des modules (fichiers `.mli`) et ajouter une nouvelle annotation, $\xrightarrow{export}$, permettant d'indiquer explicitement les éléments d'interface.

L'ajout de ces annotations est très facile : il suffit de parcourir le fichier d'interface et, pour chaque élément, utiliser son nom complètement qualifié pour ajouter une relation $\xrightarrow{export}$ entre le module ou la classe et l'élément en question. La recherche d'éléments inutilisés est alors modérée par ces arcs $\xrightarrow{export}$.

3.6. A propos des classes

Les exemples ci-dessus n'utilisent pas les classes, mais la construction de sommets dans le graphe pour les classes, méthodes et variables d'instance est intuitive : une classe contient (relation $\xrightarrow{contain}$) des méthodes et variables d'instances, ainsi que des valeurs ou modules locaux comme éléments "non exportables". Les relations d'héritage donnent des arcs $\xrightarrow{inherit}$ qui correspondent plus ou moins à la relation $\xrightarrow{include}$ pour les modules.

Le problème posé par les objets est la résolution dynamique des appels de méthodes. En effet, considérons le morceau de code suivant :

```
class c1 = object method m = 15 end;;
class c2 = object method m = 10 end;;
let obj =
  match Random.int 5 with
    0 -> new c1
  | 1 -> new c2
  | n -> f n (* en supposant f définie et renvoyant un objet du bon type *)
;;
print_int obj#m;;
```

Il n'est pas possible statiquement de déterminer si la méthode `m` invoquée dans l'expression `obj#m` est celle de la classe `c1`, celle de la classe `c2` ou bien celle d'un objet renvoyé par la fonction `f`.

Nous nous "contentons" donc de considérer que toutes les méthodes sont référencées et ne sont jamais inutilisées. Pour cette raison, lorsqu'une classe c_2 hérite d'une classe c_1, nous considérerons que la classe c_1 est utilisée, même si aucune de ses variables d'instances et méthodes n'est utilisée. La relation $\xrightarrow{inherit}$ indiquera donc une utilisation. Cela diffère de la relation $\xrightarrow{include}$ utilisée pour les modules, qui n'induit pas une utilisation du module inclus puisqu'on peut vérifier par le graphe si les éléments du module inclus sont utilisés ou non, et donc si le module lui-même est utilisé.

4. Utilisations du graphe de dépendances

Voyons maintenant comment exploiter le graphe construit dans la section précédente. Nous définissons tout d'abord un langage permettant de sélectionner dans le graphe les sommets (éléments) d'après certaines contraintes. Ensuite, nous utilisons ce langage pour extraire les informations qui nous intéressent (cf. section 2.1).

4.1. Langage de sélection de sommets dans le graphe

Nous souhaitons pouvoir sélectionner les éléments par leur nom et leur type d'une part, par les relations qu'ils ont avec d'autres éléments d'autre part.

Nous définissons donc un langage permettant de construire des filtres à appliquer sur le graphe. Un filtre permet de ne conserver qu'un ensemble de sommets et les arcs entre ces sommets. Les arcs concernant des sommets qui ne sont pas conservés sont retirés.

La notation $< exp >_k$ nous permettra de décrire la sélection des éléments dont le nom correspond à l'expression $< exp >$ et dont le type est l'un de ceux indiqués dans k. Par exemple, $< F.* >_v$ permet d'extraire toutes les valeurs ($k = v$) ayant un nom pleinement qualifié correspondant à l'expression $< F.* >$, c'est-à-dire toutes les valeurs du module `F`. k peut prendre les valeurs suivantes : v (valeur), M (module), C (classe), m (méthode), i (variable d'instance), t (type), f (champ de type).

Les opérations &, |, − et ! seront respectivement les opérations classiques d'intersection, d'union, de différence et de complément sur les ensembles. Ainsi $! < F.* >_v$ donnera tous les éléments analysés sauf les valeurs du module `F`.

Enfin, les contraintes de relations seront exprimées de la façon suivante :

- $f \xrightarrow{d}$: sélectionne les éléments atteignables par la relation de dépendance d, depuis au moins un des éléments sélectionnés par le filtre f.
- $\xrightarrow{d} f$: sélectionne les éléments permettant d'atteindre, par la relation de dépendance d, au moins un des éléments sélectionnés par le filtre f.

La relation de dépendance permet d'indiquer :

- le type d'arc à prendre en compte, par exemple les arcs "Contain" et "Use" dans $\xrightarrow{contain,use}$,
- la longeur de chemin, soit par un entier, soit par "+" pour indiquer que la longeur du chemin doit être ≥ 1. Exemple : $\xrightarrow{contain,use\{+\}}$.
- un filtre supplémentaire permettant de sélectionner des éléments par lesquels les chemins intervenant dans la recherche d'éléments ne doivent pas passer. Ainsi, le filtre

$$\xrightarrow{contain,use\{+\}/<F.M.*>} < F.f >_v$$

  sélectionne les éléments permettant d'atteindre, par des arcs "Use" ou "Contain", la valeur `F.f`, sans passer par les éléments du module `F.M`.

Dans la suite, par soucis de concision, nous utiliserons la notation $e \rightarrow$ avec e désignant un élément ou un groupe d'éléments du graphe, alors que e devrait désigner un filtre renvoyant l'unique élément ou le groupe d'éléments e.

4.2. Recherche de code inutile

La recherche du code potentiellement inutile, c'est-à-dire les éléments inutiles (car non utilisés et non utilisables), revient à définir les filtres permettant de sélectionner ces éléments dans le graphe construit dans la section 3.

Nous considérons qu'un élément e est inutile si :

- aucun autre élément en dehors des éléments "sous e" ne l'utilise,

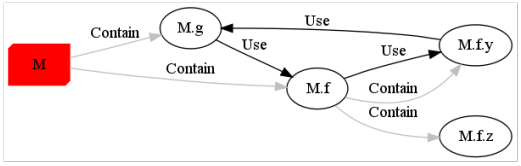


FIGURE 6 – Graphe obtenu sur un exemple de fonctions récursives.

– e ne contient que des éléments inutiles.

La notion d'élément "sous e" permet de traiter les cas de récursion, que cela soit pour les définitions de valeurs ou de classes. Considérons le code suivant :

```
let rec f x =
  let z = 1 in
  let y = g (x - 2) in
  y + 1
and g x = if x <= 1 then x else f (x + 1);;
```

La figure 6 montre le graphe produit. Les éléments `M.g`, `M.f.z`, `M.f.y` et `M.f` sont "sous `M.f`", c'est-à-dire que le fait que l'un d'eux utilise `M.f` ne permet pas de dire que `M.f` est utilisé, puisque les utilisations forment alors un cycle.

On définit donc les éléments "sous e" de la façon suivante :

$$Sub(e) = e \xrightarrow{export,contain,inherit,include,create\{+\}} |(e \xrightarrow{use,alias\{+\}} \& \xrightarrow{use,alias\{+\}} e).$$

Un élément e est utile si au moins un autre élément du graphe utilise e et ne fait pas partie de $Sub(e)$. Comme chaque élément est contenu dans un autre élément (à part les modules racines), nous ne nous intéresserons pas à la relation $\xrightarrow{contain}$. De plus, la relation $\xrightarrow{include}$ n'induit pas non plus l'utilisation. Les éléments utilisant e seront donc définis par le filtre suivant :

$$Ref(e) = \xrightarrow{use,create,export,inherit,alias\{+\}} e\ .$$

Cependant, nous devons retirer de cet ensemble les éléments qui sont des aliases de e, car il ne suffit pas que `f` $\xrightarrow{alias}$ `e` pour que e soit utilisé. Il faut qu'un autre élément g utilise cet alias :

$$g \xrightarrow{k,k\neq contain,include,alias} f \xrightarrow{alias} e.$$

On peut généraliser cet exemple au cas où une chaîne d'aliases existe de f vers e, c'est-à-dire que l'on souhaite traiter les cas $f \xrightarrow{alias\{+\}} e$. En d'autres termes, on s'autorise la recherche d'éléments utilisant e par l'intermédiaire d'aliases, mais on considère qu'un alias de e n'utilise pas e. Les aliases de e sont définis ainsi :

$$A(e) = \xrightarrow{alias\{+\}} e.$$

On peut finalement définir $U(e)$, indiquant si e est utile, de la façon suivante :

$$U(e) = card(Ref(e) - A(e) - Sub(e)) > 0.$$

Un élément qui n'est pas utilisé peut tout de même ne pas être inutile, s'il contient lui-même au moins un élément qui n'est pas inutile. C'est souvent le cas d'un module, dont on référence des

éléments qu'il contient, mais pas forcément le module lui-même. Il est pourtant utile car l'un de ses éléments est utilisé *en dehors* du module.

Pour chaque élément non utilisé, on fera un calcul supplémentaire pour savoir si tous ses éléments sont inutiles. Cependant, ce test d'utilité de chaque sous-élément doit se faire dans le contexte de l'élément conteneur que l'on teste. En effet, prenons l'exemple suivant :

```
module M = struct
  let f x = x + 1
  let g x = f x + 1
end;;
```

L'élément `M.f` est utile, puisque `M.g` y fait appel. Cependant, lors de l'analyse de `M` pour savoir si ce module contient des éléments utiles, nous devons nous intéresser aux utilisations de ses sous-éléments *en dehors* de `M`. Nous reprenons donc les définitions ci-dessus pour pouvoir les utiliser non pas dans le contexte de l'élément e mais dans celui de son conteneur C. Les éléments "sous e" dans le contexte de C sont alors :

$$Sub(e, C) = C \xrightarrow{export,contain,inherit,include,create\{+\}} |(e \xrightarrow{use,alias\{+\}} \& \xrightarrow{use,alias\{+\}} e)$$

Pour la recherche des éléments référençant e, on ajoute une contrainte, celle de ne pas utiliser l'élément C dans les chemins possibles pour atteindre les éléments en question :

$$Ref(e, C) = \xrightarrow{use,create,export,inherit,alias\{+\}/C} e$$

Ainsi, nous définissons $Cont(e)$ indiquant si l'élément e contient au moins un élément utile (en dehors de e) :

$$Cont(e) = \exists e', e \xrightarrow{contain\{+\}} e' \& \, card(Ref(e', e) - A(e') - Sub(e', e)) > 0$$

Enfin, différents filtres supplémentaires interviennent :
- les méthodes ne sont jamais inutiles (cf. 3.6),
- il est possible et utile de supposer certains éléments comme utiles. Examinons le cas suivant :
  `module P = struct type t = int let compare = Pervasives.compare end ; ;`
  `module MyMap = Map.Make(P) ; ;`
  Si le code du module `Map.Make` n'est pas connu, `P.t` et `P.compare` ne seront jamais référencés.
- pour la même raison, on peut considérer les modules vides comme utiles.

4.3. Recherche des champs de types inutiles

Avec notre graphe des dépendances, la recherche des champs de types non "créés" ou non "lus" est facilitée. Ainsi, les champs de types non créés sont obtenus par le filtre suivant :

$$< * >_f \, \& \, ! < * > \xrightarrow{create} .$$

Ce filtre s'interprète ainsi : l'intersection de l'ensemble des champs de types ($< * >_f$) et de l'ensemble des éléments qu'aucun élément ne crée ($! < * > \xrightarrow{create}$).

De façon symétrique, les champs de types non lus sont obtenus avec le filtre

$$< * >_f \, \& \, ! < * > \xrightarrow{use}$$

où l'on a cette fois l'intersection avec les éléments qu'aucun autre élément n'utilise.

4.4. Réduction du graphe

Si l'on ne s'intéresse qu'aux éléments nommables dans l'interface, on pourra réduire le graphe à ces éléments. Cela permet d'avoir plus rapidement des résultats sur ces éléments et de faciliter l'écriture de filtres.

La réduction du graphe se fait récursivement. Les éléments à réduire sont par exemple les valeurs (qui ne doivent plus contenir de sous-éléments), les modules non nommés (contructions `struct ... end` sans nom), etc., c'est-à-dire tout ce qui ne peut être exporté explicitement dans un fichier `.mli`.

Le principe est le suivant : pour chaque élément du graphe, on descend dans les éléments qu'il contient, que l'on tente de réduire à leur tour. Ensuite, s'il faut réduire l'élément en cours, on supprime chaque arc entre cet élément e et un autre élément f pour le remplacer par un arc de même type entre le conteneur de e et l'élément f.

4.5. Autres filtres utiles

Le langage de filtres permet d'effectuer dans le graphe d'autres recherches que celles évoquées ci-dessus. Par exemple, il peut être utile de rechercher les valeurs qui initialisent un certain champ `M.myrec.myfield` sans l'avoir lu. On utilisera ce filtre sur le graphe réduit :

$$\xrightarrow{create} < M.myrec.myfield > \& \, ! \xrightarrow{use} < M.myrec.myfield > \& \; < * >_v \; .$$

De même, il est parfois utile pour comprendre un code de voir ce qui lie deux fonctions, c'est-à-dire par l'intermédiaire de quels éléments une fonction `M1.f` utilise une fonction `M2.g` (bien sûr sans tenir compte des possibilités de passage de fonctions en paramètres, cf. la remarque en fin de section 3.2). Le filtre suivant (sur le graphe réduit) permet de sélectionner les éléments situés "entre" ces deux fonctions, et les deux fonctions elles-mêmes :

$$(< M1.f >_v \xrightarrow{use,alias\{+\}} \& \xrightarrow{use,alias\{+\}} < M2.g >_v) | < M1.f >_v \; | < M2.g >_v \; .$$

4.6. Utilisation de statistiques

Toujours dans l'optique de faciliter la compréhension d'un gros code existant, on peut imaginer quelques traitements statistiques sur ce graphe.

Nous pouvons par exemple chercher les éléments corrélés, en nous intéressant aux éléments référencés dans le même contexte, comme les valeurs utilisées dans les mêmes valeurs.

Pour ce faire, nous utilisons le graphe réduit, car il s'agit ici de comprendre les liens entre éléments au niveau de l'application en nous affranchissant du style. Si nous utilisions le graphe non réduit, les deux expressions

```
let x = let tmp = f x + 1 in let tmp2 = tmp + 3 in tmp2 * 4;;
let x = (f x + 4) * 4;;
```

ne donneraient pas le même résultat car dans le premier cas la fonction `f` serait utilisée par l'élément `tmp`, alors que dans le deuxième cas elle serait utilisée directement par `x`.

Pour déterminer le dégré de corrélation de deux éléments, nous utilisons le calcul de distance présent dans le test d'indépendance du χ^2 avec deux variables de Bernoulli (une par élément) prenant chacune deux valeurs possibles, "référencé" et "non référencé". Pour chaque élément e différent de e_1 et e_2, nous regardons si au moins un arc existe entre e et e_1 d'une part, et entre e et e_2 d'autre part. On ne s'intéresse pas aux arcs $\xrightarrow{contain}$, $\xrightarrow{include}$ou $\xrightarrow{export}$dans ce cas, car ils ne reflètent pas une utilisation. Selon l'existence des arcs, on incrémente la case correspondante dans le tableau suivant :

	e_1 référencé	e_1 non référencé	
e_2 référencé	$c_{0,0}$	$c_{1,0}$	$c_{*,0} = c_{0,0} + c_{1,0}$
e_2 non référencé	$c_{0,1}$	$c_{1,1}$	$c_{*,1} = c_{0,1} + c_{1,1}$
	$c_{0,*} = c_{0,0} + c_{0,1}$	$c_{1,*} = c_{1,0} + c_{1,1}$	$t = c_{*,0} + c_{*,1}$

Par exemple si un arc $e \rightarrow e_1$ existe mais qu'il n'y a pas d'arc $e \rightarrow e_2$, on incrémente $c_{0,1}$.

Ensuite, nous calculons la distance (par rapport à l'indépendance) du test du χ^2 sur cette matrice et conservons cette valeur :

$$v = \sum_{i=0,1} \sum_{j=0,1} \frac{(c_{i,j} - u_{i,j})^2}{u_{i,j}}$$

avec $u_{i,j} = \frac{c_{i,*}.c_{*,j}}{t}$.

Nous faisons ce calcul pour tous les éléments (ou éventuellement seulement ceux dont nous recherchons les éléments avec lesquels ils sont corrélés).

Pour un élément e, nous pouvons alors classer les éléments avec lesquels l'utilisation de e semble corrélée, selon les distances v calculées pour e et chacun des autres éléments.

Il est possible de restreindre les types d'arcs auxquels on s'intéresse. On peut ainsi découvrir des liens entre des champs de types de données en regardant seulement les arcs $\xrightarrow{create}$, de façon à mettre en évidence la corrélation des affectations à des champs différents.

Nous ne faisons pas un test du χ^2 mais utilisons seulement la formule de distance par rapport à l'indépendance pour classer, pour chaque élément e, les autres éléments par dégré de corrélation avec e. En effet, les conditions requises pour appliquer un test du χ^2 ne sont pas a priori réunies puisque, par exemple sur quatre éléments e_1, e_2, e_3, e_4, cela reviendrait à faire le test d'indépendance de e_1 et e_2 en considérant e_3 et e_4 indépendants, puis à considérer ultérieurement e_1 et e_2 indépendants pour tester l'indépendance de e_3 et e_4.

Cependant, même sans faire le test et en n'utilisant que la distance par rapport à l'indépendance, les résultats obtenus sur L montrent sans ambiguité des corrélations entre éléments dont les utilisations sont effectivement corrélées.

4.7. Contrôles automatiques

Une autre utilisation du graphe est le contrôle régulier du code source selon des contraintes définies par les développeurs.

On peut imaginer par exemple la situation suivante. Un module `M` définit un type `t`. On souhaite que les valeurs de ce type ne soient pas construites ou filtrées par d'autres modules, sauf un module de debogage `Debug`.

OCaml ne permet pas de déclarer une sorte de module "ami" de `M`, l'interface offerte par `M` étant la même pour tous les autres modules.

Une solution est donc de ne pas rendre le type `M.t` abstrait mais d'ajouter un contrôle pour s'assurer que les champs du type `M.t` ne sont lus et créés que dans les modules `M` et `Debug`.

Cela est fait facilement à l'aide du filtre suivant qui donne la liste des éléments accédant aux champs de `M.t` mais n'étant pas définis dans les modules `M` et `Debug` :

$$\xrightarrow{use,create} < M.t.* >_f \ \& \ !(< M.* > \mid < Debug.* >)$$

Par ailleurs, les analyses de recherche de code et de champs de types inutiles peuvent être relancées régulièrement pour détecter au plus tôt ces anomalies.

5. Implémentation et résultats

5.1. Implémentation

Cette analyse par graphe de dépendances est implémentée dans le logiciel libre Oug [1]. Il s'agit d'un outil en ligne de commande permettant l'analyse des fichiers `.ml` et `.mli` ainsi que le stockage et la lecture de graphes pour effectuer des analyses sur plusieurs parties du code sans reconstruire les graphes à chaque fois. Le langage de filtres est implémenté et permet à l'utilisateur de spécifier, à l'aide d'une syntaxe adéquate, des filtres pour effectuer des recherches dans le graphe, comme indiqué en section 4. La recherche d'éléments inutiles est disponible directement par une option (ce n'est pas à l'utilisateur de construire et d'appliquer les filtres pour chaque élément), de même que l'analyse de corrélation. Enfin, diverses sorties au format Graphviz sont disponibles.

Une démonstration est accessible à l'adresse `http://yquem.inria.fr/~guesdon/demo.x`.

Nous donnons maintenant quelques détails d'implémentation.

Les éléments (modules, classes, valeurs, types, etc.) sont stockés dans un *tableau des éléments* et ont ainsi un identifiant unique qui est leur indice dans ce tableau. Même si tous les éléments ne peuvent pas forcément être nommés de façon unique, l'utilisation des identifiants évite toute ambiguïté. Pour chaque élément, on conserve donc :

- son nom pleinement qualifié, pas forcément unique (par exemple, on peut avoir deux valeurs appelées `Module.f`),
- la localisation de sa définition, quand elle est disponible dans le *typedtree*, ceci afin de pouvoir l'afficher et indiquer sans ambiguïté à l'utilisateur la position de l'élément en question,
- le type d'élément : module, classe, valeur, méthode, variable d'instance, type ou champ de type.

Le graphe contient deux tableaux, permettant d'associer à chaque élément la liste des arcs dont il est l'origine d'une part, dont il est la destination d'autre part. Ces listes sont des paires (identifiant d'élément, annotation de dépendance).

L'outil prend en paramètre une liste de fichiers sources OCaml. Ces fichiers doivent être donnés dans leur ordre de dépendance (ceux ne dépendant de rien en premiers), afin qu'un élément du graphe soit construit après les éléments dont il dépend, pour pouvoir créer les arcs de dépendance. Il est possible de construire incrémentalement le graphe en utilisant des options de stockage et de relecture d'un graphe et de l'environnement.

L'outil se lie avec le compilateur OCaml, afin d'en utiliser le *typedtree*. Cela permet d'avoir les "open" déjà résolus (sans avoir à rejouer la mécanique de recherche et d'ouverture de fichiers signatures) et les contraintes de types nécessaires à la gestion d'un environnement utilisé lors de l'analyse. En effet, les noms "résolus" dans le *typedtree* ne sont pas les noms pleinement qualifiés. Les sommets du graphe sont créés en parcourant les *typedtree* des différents modules à analyser, l'environnement permettant d'une part de résoudre les identifiants rencontrés (en gérant la portée lexicale) et d'autre part d'associer un nom pleinement qualifié de module, classe, valeur, etc., à l'identifiant du sommet créé dans le graphe. En particulier un élément est toujours créé avant les éléments qu'il contient (relation $\xrightarrow{contain}$) et a donc toujours un identifiant inférieur à ces derniers. Par ailleurs, l'utilisation du *typedtree* permet de garantir que le programme est correctement typé et qu'il n'y a pas d'identifiant non lié. Enfin, comme indiqué dans les perspectives, les indications de types pourraient servir pour des analyses ultérieures.

Lors de l'analyse d'un fichier `.mli`, il peut y avoir plusieurs éléments possibles correspondant à un élément présent dans ce fichier. Par exemple :

Fichier .mli	Fichier .ml
`val x : int`	`let x = 3 ; ;` `let x = 4 ; ;`

1. `http://pauillac.inria.fr/~guesdon/oug.en.html`

On considèrera, comme le compilateur, que c'est l'élément créé en dernier qui est exporté, c'est-à-dire celui ayant l'identifiant le plus élevé (ici le `x` de `let x = 4`).

5.2. Résultats

Sur le code du logiciel L, une première utilisation de Oug a permis de détecter et supprimer plusieurs dizaines d'éléments inutiles, incluant des champs d'enregistrements et des constructeurs. Le code est ainsi passé de 59.600 lignes à 57.300, soit une diminution de 3.85% de la taille du code. Certains éléments ont cependant été conservés en attendant de savoir si certaines fonctionnalités seraient à nouveau intégrées ou non. De plus, du code de debogage actuellement inutilisé n'a pas été supprimé non plus (commenter ou supprimer, telle est la question).

Lors des évolutions d'architecture prévues pour le logiciel, il est probable que Oug permettra de détecter et supprimer encore d'autres parties devenues inutiles.

6. Conclusion et perspectives

Nous avons défini la construction d'un graphe de dépendances pour du code OCaml et différentes façons d'exploiter ce graphe, notamment dans un contexte de reprise d'un logiciel existant. Cette analyse est implémentée dans un logiciel libre, Oug.

Les modules récursifs ne sont pas encore complètement traités par l'outil. La difficulté consiste à créer des sommets référençant des sommets inconnus au moment de l'analyse. Deux passes d'analyse seront nécessaires. Le problème ne se pose pas pour les classes car leurs éléments ne sont pas référencés depuis l'extérieur des classes (en l'absence d'analyse de flots de données).

Une piste pour enrichir le graphe serait la gestion des paramètres et l'ajout de liens entre ces paramètres et les autres éléments. A première vue, cela entraînerait la nécessité de créer un sommet dans le graphe pour chaque expression du programme analysé. On se trouverait alors au bord de l'analyse de flots de données.

Une autre piste de réflexion est l'ajout d'autres traitements statistiques pour faire ressortir des informations pertinentes pour la compréhension du programme à reprendre. Les analyses statistiques sont utiles sur des données incomplètes ou contenant des erreurs. Cela correspond pour notre cas à l'analyse d'un gros logiciel dont on cherche à comprendre l'organisation générale malgré un manque d'homogénéité dans sa structure. Pour cela, l'ajout d'un type d'arc reflétant le type des valeurs dans le graphe serait sans doute utile.

Enfin, le graphe pourrait être exploité pour détecter des *code smells* [5], en définissant certains critères pour identifier automatiquement des parties douteuses du code.

Références

[1] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman, *Compilateurs - Principes, techniques et outils, 2ème édition*, Pearson Education, 2007.

[2] Paul Anderson, Thomas Reps, Tim Teitelbaum, *Design and implementation of a fine-grained software inspection tool*, IEEE Transactions on software engineering, Volume 29, Issue 8 (Aug. 2003)

[3] Matthias Blume, *Dependency analysis for Standard ML*, ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 21, Issue 4 (July 1999)

[4] Xavier Leroy et al., *The Objective Caml system release 3.11*, INRIA, 2008

[5] Eva van Emben, Leon Moonen, *Java quality assurance by detecting code smells*, Ninth Working Conference on Reverse Engineering, 2002.

Faire bonne figure avec MLPOST

R. Bardou[1] & J.-C. Filliâtre[1] & J. Kanig[1] & S. Lescuyer[1]

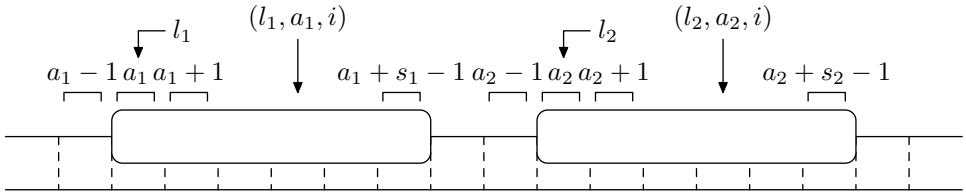
1: ProVal / INRIA Saclay – Île-de-France
91893 Orsay Cedex, France
LRI / CNRS – Université Paris Sud
91405 Orsay Cedex, France
{bardou,filliatr,kanig,lescuyer}@lri.fr

Résumé

Cet article présente MLPOST, une bibliothèque OCaml de dessin scientifique. Elle s'appuie sur METAPOST, qui permet notamment d'inclure des fragments LaTeX dans les figures. OCaml offre une alternative séduisante aux langages de macros LaTeX, aux langages spécialisés ou même aux outils graphiques. En particulier, l'utilisateur de MLPOST bénéficie de toute l'expressivité d'OCaml et de son typage statique. Enfin MLPOST propose un style déclaratif qui diffère de celui, souvent impératif, des outils existants.

1. Introduction

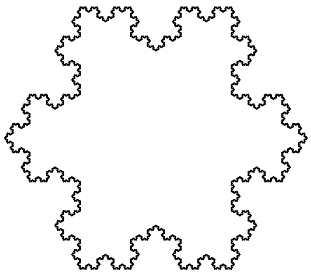
Lors de la rédaction de documents à nature scientifique (articles, cours, livres, etc.), il est très souvent nécessaire de réaliser des figures. Ces figures permettent d'agrémenter le texte en illustrant aussi bien les objets dont il est question dans le document que les liens qui existent entre eux et facilitent ainsi leur compréhension. Elles sont donc un composant fondamental au caractère didactique de tels documents, mais leur réalisation est souvent fastidieuse. En particulier, il est souvent nécessaire d'y inclure des éléments mis en forme par LaTeX (formules, etc.), ce que bon nombre de logiciels de dessin ne permettent pas. Ainsi, on peut souhaiter réaliser un schéma tel que celui-ci



en attachant de l'importance au fait que des expressions comme $a_1 + s_1 - 1$ apparaissent exactement comme dans le corps du document. Il existe plusieurs familles d'outils pour réaliser des figures à intégrer dans un document LaTeX :

- des interfaces graphiques disposant d'une sortie LaTeX, telles que Dia [12] ou Xfig [14] ;
- des bibliothèques LaTeX, telles que PSTricks [8] ou encore TikZ [15] ;
- des outils externes en ligne de commande, tel que METAPOST [10].

Chaque famille a ses avantages et ses inconvénients. Les interfaces graphiques sont les plus accessibles, notamment pour un placement rapide et intuitif des différents éléments de la figure, mais l'intégration de texte mis en forme par LaTeX est délicate. Dans le cas de Xfig et de Dia, la taille des éléments LaTeX n'est pas connue lors de l'édition de la figure ; en outre, dans le cas de Dia, l'intégration de LaTeX dans une figure nécessite de l'exporter sous forme de macros TikZ et d'éditer le résultat.



```
vardef koch(expr A,B,n) =
  save C; pair C; C = A rotatedaround(1/3[A,B], 120);
  if n>0:
    koch( A,        1/3[A,B], n-1);
    koch( 1/3[A,B], C,        n-1);
    koch( C,        2/3[A,B], n-1);
    koch( 2/3[A,B], B,        n-1);
  else:
    draw A--1/3[A,B]--C--2/3[A,B]--B;
  fi;
enddef;
z0=(4cm,0); z1=z0 rotated 120; z2=z1 rotated 120;
koch( z0, z1, 4 ); koch( z1, z2, 4 ); koch( z2, z0, 4 );
```

FIGURE 1 – Exemple de figure METAPOST

Les bibliothèques LaTeX telles que PSTricks ou TikZ offrent l'intégration la plus naturelle avec LaTeX. En particulier, elles permettent de combiner arbitrairement éléments graphiques et textes LaTeX « comme ceci ». En revanche, elles demandent d'apprendre un certain nombre de macros et de notations et souffrent surtout des défauts inhérents à LaTeX :

- des erreurs détectées uniquement à l'interprétation, peu claires et parfois mal localisées ;
- un langage *de programmation* peu commode (syntaxe obscure, absence de typage, code difficile à structurer).

Ces inconvénients sont notamment un frein au développement de bibliothèques de haut niveau au dessus de ces langages ainsi qu'à la réutilisation de figures.

METAPOST se présente comme une alternative à ces bibliothèques LaTeX, en proposant un langage de programmation à part entière spécialisé dans la construction de figures contenant des éléments LaTeX. Il permet notamment de manipuler symboliquement la taille et la position de ces éléments et de les relier de manière implicite par des équations. En revanche, le langage de METAPOST s'inspire de celui de METAFONT [11] et présente, à l'exception de la syntaxe, les défauts soulevés ci-dessus. La figure 1 donne un exemple de programme/figure réalisé avec METAPOST.

Une alternative séduisante aux solutions précédentes consiste à utiliser un langage de programmation existant. Ainsi l'utilisateur n'a pas à apprendre un langage spécialisé et il bénéficie d'autre part de tous les avantages d'un langage de programmation moderne : erreurs détectées à la compilation, types de données complexes, structuration, etc. Toute la difficulté réside alors dans la manipulation des éléments LaTeX, notamment la prise en compte de leur taille dans l'élaboration de la figure. Si on considère la famille des langages fonctionnels, on peut citer au moins deux exemples de telle intégration :

- mlPicTeX [4] est [1] un ensemble de macros LaTeX permettant d'inclure du code Caml Light arbitraire dans un document LaTeX. Ce code s'appuie sur une bibliothèque de dessin PostScript [13] et peut faire référence à des éléments LaTeX, ainsi qu'à leur taille.
- *functional* METAPOST [9] est une bibliothèque Haskell [1] produisant du code METAPOST. C'est une approche légère qui réutilise les capacités graphiques de METAPOST et substitue Haskell au langage de programmation de METAPOST.

Cet article présente MLPOST, un outil qui adopte l'approche de *functional* METAPOST en utilisant OCaml [2] comme langage hôte. La première partie de l'article présente les choix de conception de MLPOST à travers un certain nombre d'exemples. La seconde partie détaille ensuite l'architecture logicielle de MLPOST. MLPOST est librement distribué à l'adresse `http://mlpost.lri.fr`. Toutes les figures de cet article ont été faites avec MLPOST, à l'exception des exemples pour METAPOST et TikZ.

1. À notre connaissance, mlPicTeX n'est plus distribué.

2. Principes et exemples

2.1. Principes

Boîtes. Les briques de base de MLPOST sont les *boîtes* : une boîte est un moyen d'encapsuler n'importe quel élément de dessin au sein d'un contour, qui peut être effectivement tracé ou non. On peut construire la boîte vide, des boîtes avec du LaTeX arbitraire, etc. Ces boîtes peuvent ensuite être manipulées : imbrication arbitraire, placement à une position précise, alignement de plusieurs boîtes, flèches reliant plusieurs boîtes entre elles, création de tableaux, etc. Plusieurs boîtes peuvent aussi être regroupées au sein d'une seule afin de pouvoir les déplacer ensemble. L'exemple suivant montre deux boîtes simples, la deuxième étant déplacée un centimètre vers la droite en utilisant la fonction `shift`.

LaTeX

```
[ Box.draw (Box.tex "\\LaTeX");
  Box.draw (Box.shift (Point.pt (cm 1., zero)) (circle (empty ()))) ]
```

Une figure MLPOST est simplement une liste de commandes de dessin. Ci-dessus, elle est réduite à deux occurrences de `Box.draw`, la commande qui dessine une boîte.

Placement relatif. Un principe que nous avons suivi lors de la conception de MLPOST est de favoriser un placement relatif des objets plutôt qu'absolu. Ceci permet d'obtenir des figures plus robustes. En effet, imaginons que l'on veuille placer une boîte A à *droite* d'une boîte B. Une première possibilité serait de spécifier les positions approximativement, par exemple en donnant les abscisses 0 cm pour A et 2 cm pour B. Cependant, si on change d'avis sur le contenu de A et que la taille de cette boîte change, A risque alors de se superposer à B. Il faut alors replacer toutes les boîtes de la figure manuellement. Pour éviter ça, MLPOST propose diverses méthodes pour placer les boîtes *les unes par rapport aux autres*. On gagne alors du temps lors de la création et lors des modifications de la figure. L'exemple suivant utilise l'alignement horizontal `hbox`, où l'argument optionnel `padding` permet de spécifier l'espacement horizontal entre deux boîtes :

LaTeX

```
[ Box.draw (Box.hbox ~padding:(cm 1.)
                     [Box.tex "\\LaTeX"; circle (empty ())]) ]
```

Nous revenons plus en détail sur les boîtes et leur implémentation dans la section 3.2.

Persistance. Un autre choix que nous avons fait est celui de la persistance [7] : lorsqu'un attribut d'une boîte (positionnement, couleur, etc.) est modifié, on obtient une nouvelle boîte, identique à la première sauf en l'attribut changé. En faisant le choix de structures de données persistantes, nous permettons à l'ancienne boîte, avec ses attributs inchangés, d'être préservée et encore accessible. Ainsi, on peut réutiliser plus facilement une boîte à plusieurs endroits du dessin, avec des attributs différents. Dans l'exemple suivant, on a utilisé trois instances de la même boîte `b`, dont le contour est tracé pour la deuxième.

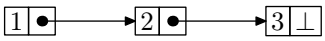
LaTeX
LaTeX
LaTeX

```
let b =
  Box.hbox ~padding:(cm 1.) [Box.tex "\\LaTeX"; circle (empty ())] in
[ Box.draw (Box.vbox [b; set_stroke Color.black b; b]) ]
```

2.2. Exemples

Dans cette section, nous montrons quelques applications immédiates des boîtes de MLPOST.

Représentation de la mémoire. Un besoin récurrent lorsque l'on enseigne l'algorithmique ou les concepts liés à un langage de programmation consiste à illustrer la structure des données en mémoire par des schémas de la forme



Deux éléments sont nécessaires : le dessin des blocs d'une part et le dessin des pointeurs d'autre part. Pour réaliser les blocs, on utilise la fonction `Box.hblock` qui aligne des boîtes horizontalement, leur donne une hauteur commune et trace leur contour. Voici un exemple :

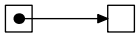
```
let b = Box.hblock ~pos:`Bot [Box.tex "a"; Box.tex "b"; Box.tex "C"] in
[ Box.draw b ]
```

À l'aide de `Box.hblock`, on peut facilement écrire une fonction `cons` qui construit un bloc de taille 2, dont le premier élément contient un texte LaTeX arbitraire `hd` et le second est soit vide, soit le symbole ⊥, selon la valeur du booléen `tl` :

```
let cons hd tl =
  let p1 = Box.tex ~name:"hd" hd in
  let p2 = Box.tex ~name:"tl" (if tl then "" else "\\ensuremath{\\bot}") in
  Box.hblock [p1; p2]
```

L'argument optionnel `~name` permet de nommer les sous-boîtes, de manière à pouvoir y accéder facilement par la suite.

Écrivons maintenant une fonction `pointer_arrow` pour matérialiser les pointeurs. Il s'agit de tracer une flèche entre deux boîtes données `a` et `b`. Pour cela, on commence par construire un chemin `p` reliant les centres de `a` et `b`, que l'on tronque à l'endroit où il intersecte le bord de la boîte `b` (avec la fonction `Path.cut_after`). Puis on trace l'origine de la flèche à l'aide d'un chemin réduit à un point (le centre de `a`) et la flèche proprement dite à l'aide du chemin `p`.



```
let pointer_arrow a b =
  let p = pathp [Box.ctr a; Box.ctr b] in
  let p = Path.cut_after (Box.bpath b) p in
  let pen = Pen.scale (bp 4.) Pen.circle in
  Command.draw ~pen (pathp [Box.ctr a]) ++ draw_arrow p
```

On utilise ici le symbole infixe `++` qui permet de concaténer des commandes de dessin. On utilise d'autre part `bp`, qui est une unité propre à METAPOST, proche du point PostScript.

Nous avons maintenant tous les éléments nécessaires pour écrire une fonction `draw_list` qui prend en argument une liste OCaml de fragments LaTeX et réalise l'illustration correspondante. On commence par construire les différents blocs constituant la liste :

```
let draw_list l =
  let rec make = function
    | [] → []
    | [x] → [cons x false]
    | x :: l → cons x true :: make l in
```

Puis on aligne ces blocs avec `Box.hbox`, en insérant de l'espace horizontal avec l'option `padding` :

```
  let l = hbox ~padding:(bp 30.) (make l) in
```

Pour dessiner les pointeurs, il suffit de parcourir la liste des boîtes (qui ont été placées) et d'utiliser la fonction `pointer_arrow` précédente sur chaque paire de boîtes consécutives :

```
  let rec arrows = function
    | [] | [_] → nop
    | b1 :: (b2 :: _ as l) →
        pointer_arrow (Box.get "tl" b1) (Box.get "hd" b2) ++ arrows l in
```

On utilise ici la fonction `Box.get` qui permet de récupérer une sous-boîte par son nom. On accède ainsi aux boîtes nommées respectivement `"hd"` et `"tl"` qui ont été créées par la fonction `cons` puis encapsulées dans d'autres boîtes par les fonctions d'alignement. Enfin, on dessine les boîtes avec `Box.draw`, puis les flèches avec la fonction `arrows` :

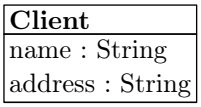
```
[ Box.draw l; arrows (Array.to_list (Box.elts l)) ]
```

On peut tester avec

```
draw_list (List.map (fun n → Printf.sprintf "$\\sqrt{%d}$" n) [1;2;3;4])
```

qui donne bien le résultat attendu :

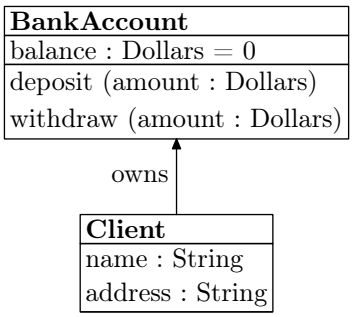
Diagrammes de classes. Avec la fonction `Box.vblock`, analogue pour l'alignement vertical de la fonction `Box.hblock` introduite ci-dessus, il est facile de dessiner des diagrammes UML. Supposons que l'on veuille dessiner des schémas de classes tels que :



Pour cela, introduisons une fonction `classblock` qui attend le nom de la classe ainsi que la liste des attributs et des méthodes.

```
let classblock name attr_list method_list =
  let vbox = Box.vbox ~pos:`Left in
  Box.vblock ~pos:`Left ~name
  [ tex ("{\\bf " ^ name ^ "}");
    vbox (List.map tex attr_list); vbox (List.map tex method_list) ]
```

Ici, le nom `name` de la classe est utilisé à la fois pour désigner le schéma dans le diagramme (l'argument labelisé `~name` de `Box.vblock`) et comme titre du schéma créé. Les attributs et les méthodes sont alignés verticalement indépendamment, puis on aligne le titre et les deux nouvelles boîtes obtenues en les encadrant [2]. On peut maintenant s'en servir pour dessiner un petit diagramme de classes :




```
let a = classblock "BankAccount"
          [ "balance : Dollars = $0$"]
          [ "deposit (amount : Dollars)";
            "withdraw (amount : Dollars)" ] in
let b = classblock "Client"
          [ "name : String"; "address : String" ] [] in
let diag = Box.vbox ~padding:(cm 1.) [a;b] in
[ Box.draw diag;
  box_label_arrow ~pos:`Left (Picture.tex "owns")
    (get "Client" diag) (get "BankAccount" diag) ]
```

Ici, on a d'abord créé deux schémas de classe avec la fonction `classblock`. Ces schémas sont ensuite alignés verticalement, et une flèche avec une étiquette est dessinée entre ces deux classes avec `box_label_arrow`. Le code pour cette figure est conceptuellement très simple, ne contient aucun placement absolu et ne dépasse pas les 15 lignes de code.

2. Notez qu'au contraire de `hbox` et `vbox`, les fonctions `hblock` et `vblock` tracent par défaut le contour de leurs sous-boîtes.

Automates. La théorie des langages est un domaine où l'on a rapidement besoin de dessiner des automates. Illustrons une façon d'utiliser MLPOST dans ce but. Nous allons définir les fonctions suivantes :

- `state` pour créer un état ;
- `final` pour transformer un état en un état final ;
- `initial` pour dessiner une flèche entrante sur un état initial ;
- `transition` pour dessiner une transition d'un état à un autre ;
- `loop` pour dessiner une transition d'un état vers lui-même.

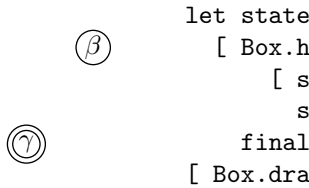
On choisit de représenter les états par des boîtes MLPOST. La fonction `state` est juste une spécialisation de la fonction `Box.tex` à un contour circulaire, définie par l'application partielle suivante :

```
let state = Box.tex ~style:Circle ~stroke:(Some Color.black)
```

Le paramètre `stroke` permet de spécifier si le contour doit être tracé et, le cas échéant, dans quelle couleur. De manière similaire, la fonction `final` est une spécialisation de la fonction `Box.box` dont le rôle consiste à rajouter un cercle autour d'une boîte :

```
let final = Box.box ~style:Circle
```

On peut déjà placer des états, finaux ou non, et les dessiner. Pour le placement, on utilise les fonctions d'alignement horizontal et vertical `Box.hbox` et `Box.vbox`. On suit donc le principe consistant à placer les objets de façon relative, les uns par rapport aux autres.



```
let states = Box.vbox ~padding:(cm 0.8)
  [ Box.hbox ~padding:(cm 1.4)
      [ state ~name:"alpha" "$\\alpha$";
        state ~name:"beta" "$\\beta$" ];
    final ~name:"gamma" (state "$\\gamma$") ] in
[ Box.draw states ]
```

On note que l'ensemble des états est lui-même une boîte, `states`, contenant les états comme autant de sous-boîtes nommées.

La fonction `initial` appose une flèche entrante à un état. Il s'agit donc d'une fonction qui prend le nom d'un état `q` et qui renvoie une commande dessinant une flèche vers `q`. On pourrait aussi renvoyer une boîte sans contour contenant `q` et la flèche, ce qui permettrait d'utiliser `initial` de la même façon que `final`. Cependant, la boîte obtenue n'aurait pas la même taille et la même forme que `q`, ce qui poserait des problèmes pour placer `q` ou pour dessiner des transitions vers ou à partir de `q`.

```
let initial (states : Box.t) (name : string) : Command.t =
  let q = Box.get name states in
  let p = Box.west q in
  Arrow.draw (Path.pathp [Point.shift p (Point.pt (cm (-0.3), zero)); p])
```

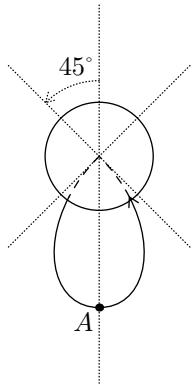
La fonction accède à la boîte `q` par son nom `name` dans la boîte `states` et détermine le point d'arrivée de la flèche avec `Box.west`. On pourrait généraliser cette fonction pour spécifier la position de la flèche.

La fonction `transition` dessine une flèche d'un état à un autre. Cette fonction prend deux arguments optionnels `outd` et `ind` pour spécifier, en degrés, la direction sortante et la direction entrante de la flèche. On doit les convertir en vecteurs directeurs pour les passer [3] à `cpath`, qui calcule un chemin allant du bord d'une boîte au bord d'une autre boîte. Ce chemin est ensuite donné à la fonction `Arrow.draw` qui trace la flèche en plaçant une étiquette `tex` à la position `pos`.

3. Nous utilisons ici une fonctionnalité d'OCaml qui permet d'accéder à des arguments optionnels sans spécifier leur valeur par défaut, sous la forme d'un type `option`. Nous passons ensuite directement ces valeurs de type `option`, en tant qu'arguments optionnels, à la fonction `cpath` par la syntaxe `cpath ?outd ?ind x y`.

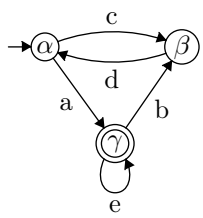
```
let transition states tex pos ?outd ?ind x_name y_name =
  let x = Box.get x_name states and y = Box.get y_name states in
  let outd = match outd with None → None | Some a → Some (vec (dir a)) in
  let ind = match ind with None → None | Some a → Some (vec (dir a)) in
  Arrow.draw ~tex ~pos (cpath ?outd ?ind x y)
```

La fonction `loop` est similaire à la fonction `transition`, mais elle doit calculer un chemin plus complexe. En effet, `cpath` appliqué à deux boîtes identiques renvoie un chemin vide et on ne peut donc pas l'utiliser. À la place, on calcule un point A suffisamment éloigné de la boîte et on trace un chemin qui part du centre, qui passe par A puis qui revient au centre. On utilise au passage la fonction `knotp` qui permet de spécifier un point avec une tangente, et `pathk` qui transforme une liste de tels points en un chemin.




```
let loop states tex name =
  let box = Box.get name states in
  let a = Point.shift (Box.south box) (Point.pt (cm 0., cm (-0.4))) in
  let c = Box.ctr box in
  let p = Path.pathk [
    knotp ~r:(vec (dir 225.)) c;
    knotp a;
    knotp ~l:(vec (dir 135.)) c;
  ] in
  let bp = Box.bpath box in
  Arrow.draw ~tex ~pos:`Bot (cut_after bp (cut_before bp p))
```

Ici encore, on pourrait généraliser cette fonction pour spécifier la position de la flèche. On peut maintenant dessiner facilement des automates en utilisant cette bibliothèque.

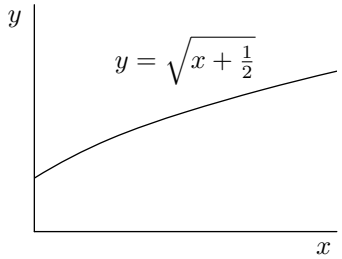



```
let automate =
  let states = ... in
  [ Box.draw states;
    transition states "a" `Lowleft "alpha" "gamma";
    transition states "b" `Lowright "gamma" "beta";
    transition states "c" `Top ~outd:25. ~ind:335. "alpha" "beta";
    transition states "d" `Bot ~outd:205. ~ind:155. "beta" "alpha";
    loop states "e" "gamma"; initial states "alpha" ]
```

2.3. Exemples utilisant des calculs en OCaml

Cette section illustre l'un des avantages de MLPOST : la capacité de dessiner directement un objet que l'on a calculé/programmé en OCaml.

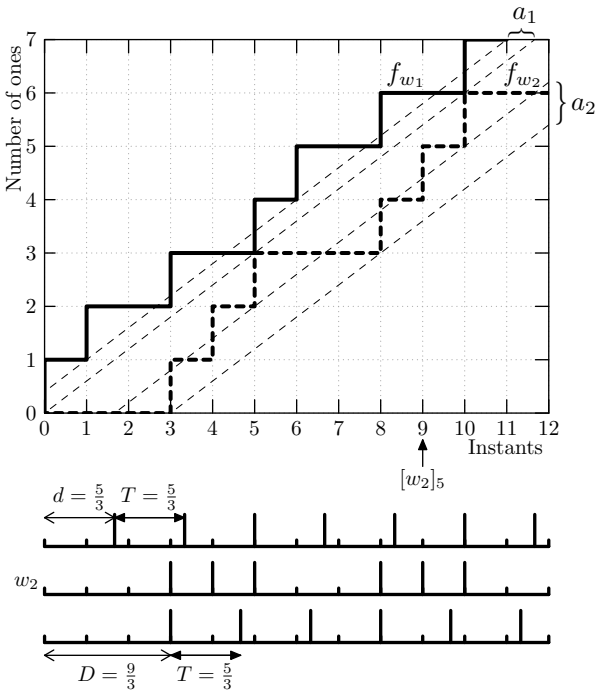
Graphe de fonction. Un exemple simple de dessin résultant d'un calcul est celui du graphe d'une fonction. MLPOST fournit un module `Plot` à cet effet. La figure suivante montre un exemple basique d'utilisation de cette extension :




```
let u = cm 1. in
let sk = Plot.mk_skeleton 4 3 u u in
let label = Picture.tex "$y=\\sqrt{x+\\frac{1}{2}}$",
            `Upleft, 3 in
let f x = sqrt (float x +. 0.5) in
let graph = Plot.draw_func ~label f sk in
[ graph; Plot.draw_simple_axes "$x$" "$y$" sk ]
```

La fonction `mk_skeleton` permet de construire un cannevas de 4 unités sur 3, qui est l'objet de base de l'extension `Plot`. Il est alors possible de dessiner un graphe de fonction et des axes au sein de ce cannevas, comme illustré ci-dessus. L'extension dispose de beaucoup d'options (tracé de la grille, affichage des abscisses et ordonnées, différents types de graphes de fonctions) qui permettent de réaliser des figures plus complexes, telle que celle décrite dans le paragraphe suivant.

Abstractions d'horloges dans un système synchrone flot-de-données. L'exemple ci-dessous, réalisé par Florence Plateau, provient d'un problème réel [5] et illustre un certain nombre des possibilités de l'extension `Plot`. Ainsi les fonctions illustrées sur cette figure ont été codées comme des fonctions OCaml standard. De plus, la ligne intermédiaire dans la partie située sous le graphe principal, et dénotée par w_2, représente les discontinuités de la fonction f_{w_2} du graphe principal. Cette ligne est calculée *directement* à partir de la fonction f_{w_2} ; si l'on décide de changer la fonction f_{w_2}, la ligne w_2 sera mise à jour automatiquement. Cela est également vrai pour certaines étiquettes de la figure, comme l'abscisse $[w_2]_5$. Ceci offre une flexiblité très intéressante lors de la phase de développement d'une telle figure.

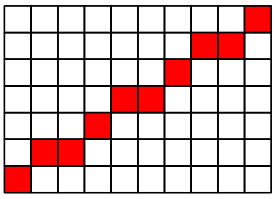


Bresenham. À titre de dernier exemple, supposons que l'on veuille illustrer l'algorithme de tracé de segment de Bresenham [3], par exemple sur le segment reliant le point $(x_1, y_1) = (0, 0)$ au point $(x_2, y_2) = (9, 6)$. Pour cela, on commence par stocker le résultat de l'algorithme dans un tableau `bresenham_data`, tel que `bresenham_data.(`x`)` donne l'ordonnée du point d'abscisse x.

```
let x2 = 9 and y2 = 6
let bresenham_data = Array.create (x2+1) 0
let () = (* remplissage du tableau a avec l'algorithme de Bresenham *) ...
```

On peut alors réaliser la figure très facilement, à l'aide de la fonction `Box.gridi` fournie par MLPOST, qui construit une matrice de boîtes alignées à partir d'une largeur, d'une hauteur et d'une fonction

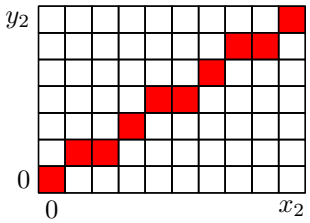
construisant la boîte (i, j), d'une manière analogue à `Array.create_matrix`.



```
let width = bp 6. and height = bp 6. in
let g = Box.gridi (x2+1) (y2+1)
  (fun i j →
    let fill = if bresenham_data.(i) = y2 - j
               then Some Color.red else None in
    Box.rect ?fill (Box.empty ~width ~height ())) in
[ Box.draw g ]
```

Les boîtes sont des boîtes vides de 6 points de côté, créées avec `Box.empty`. Pour les boîtes correspondant à des points dessinés par l'algorithme de Bresenham, on indique que la boîte doit être remplie en rouge, à l'aide de l'argument optionnel `fill`.

Pour parachever la figure, on va ajouter des étiquettes indiquant les coordonnées des deux extrémités du segment. Pour placer une étiquette à côté de la case (i, j), on récupère la boîte correspondante à l'aide de `Box.nth`, puis on récupère un point particulier de cette boîte (par exemple le point au milieu en bas avec `Box.south`), puis enfin on trace l'étiquette avec `Command.label`.




```
let bresenham =
  let width = bp 6. and height = bp 6. in
  let g = ... in
  let get i j = Box.nth i (Box.nth (y2-j) g) in
  let label pos s point i j =
    Command.label ~pos (Picture.tex s) (point (get i j)) in
  [ Box.draw g;
    label ‘Bot "0" Box.south 0 0; label ‘Bot "$x_2$" Box.south x2 0;
    label ‘Left "0" Box.west 0 0; label ‘Left "$y_2$" Box.west 0 y2 ]
```

3. Architecture logicielle

La figure 2 montre le fonctionnement de MLPOST. Tout d'abord, MLPOST est un outil de génération de fichier METAPOST sous forme de bibliothèque OCaml. À l'aide de cette bibliothèque, l'utilisateur écrit un programme qui, à l'exécution, construit un arbre de syntaxe abstraite METAPOST. Cet arbre est imprimé dans un fichier `figure.mp` qui est lu par METAPOST pour générer un ou plusieurs fichiers PostScript [4]. L'inclusion de ces figures dans un document LaTeX se fait simplement en utilisant la commande `\includegraphics` du package `graphicx`. Pour compiler le document LaTeX avec `pdflatex`, il suffit de changer l'extension des figures générées, ce qu'une option de l'outil MLPOST permet de faire facilement.

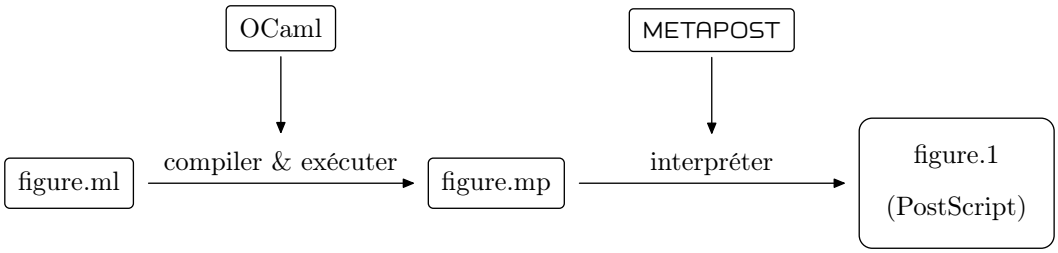


FIGURE 2 – Architecture de MLPOST

4. Ces fichiers n'ont pas le suffixe `.ps` car il leur manque l'en-tête.

Il est important de noter que le fichier METAPOST de sortie n'est pas obtenu par *compilation* du code source OCaml, mais par une *exécution* du programme qui construit un arbre de syntaxe abstraite METAPOST. Cette méthode a l'inconvénient qu'une boucle ou itération dans le programme de départ sera traduite par une suite de commandes obtenues par le déroulement de la boucle, et non par une construction de boucle du langage cible. Ceci étant dit, dans notre cas, le coût supplémentaire est faible, puisque METAPOST déroule également les boucles dans les fichiers PostScript de sortie.

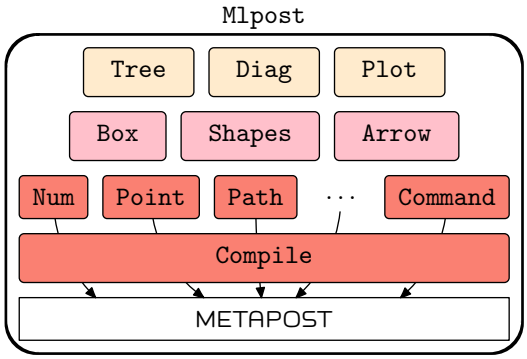


FIGURE 3 – Architecture de MLPOST

L'architecture générale de MLPOST est schématisée en figure 3. Au niveau le plus bas se trouvent les interfaces correspondant aux types primitifs de METAPOST. Ce sont ces objets qui sont *in fine* traduits en du code METAPOST, et nous les décrivons de manière plus détaillée dans la section 3.1. La couche intermédiaire contient des éléments que nous estimons être de bas niveau mais qui ne sont pas présents dans METAPOST : ils sont propres à MLPOST et ont été construits à partir de la couche inférieure. Les sections 3.2 et 3.3 reviennent plus en détail sur deux de ces modules, respectivement `Box` et `Arrow`. Enfin, on trouve au plus haut niveau des modules tels que le module `Plot` présenté dans la section précédente. L'intégralité des modules de MLPOST est empaquetée dans un module `Mlpost`, afin de ne pas polluer l'espace de noms d'OCaml.

3.1. Types primitifs de METAPOST

La couche de bas niveau de MLPOST est une interface fidèle à METAPOST. Elle comporte tous les types de base de METAPOST :

Le type numérique (module `Num`) représente des longueurs. En première approximation, ce type pourrait être assimilé au type `float` d'OCaml, mais certaines valeurs, telle que la taille d'un élément LaTeX, ne sont connues qu'à l'interprétation du fichier METAPOST. La plupart des calculs sont donc effectués de manière symbolique et le type `Num.t` doit donc être abstrait.

Le type point (module `Point`) représente des points dans l'espace à deux dimensions. Pour les mêmes raisons que les numériques, les points ne sont pas simplement des paires de flottants, mais doivent être représentés de manière symbolique. Le type `Point.t` est également utilisé pour représenter les vecteurs.

Les chemins (module `Path`) sont des lignes représentées par des courbes de Bézier. Ils sont à la base de tout dessin METAPOST. Toutes les possibilités de construction de chemin dans METAPOST ont été interfacées. On peut dessiner des lignes droites ou des lignes courbes en précisant les points de contrôle, la tension de la courbe, etc.

Les transformations (module `Transform`) permettent d'appliquer une transformation linéaire à un objet quelconque. Il est ainsi possible de déplacer des objets, les redimensionner, les faire pivoter ou encore combiner toutes ces transformations.

Les plumes (module `Pen`) permettent de choisir l'épaisseur et la forme du stylo utilisé pour dessiner les chemins.

Les figures (module `Picture`) permettent de rassembler plusieurs éléments graphiques en un seul, qu'il s'agisse d'éléments LaTeX ou de commandes de dessin arbitraires. Le type `Picture.t` permet de traiter une figure arbitrairement complexe comme un objet de base que l'on peut copier, transformer, etc. Le module `Picture` permet également de découper une figure à l'aide d'une surface décrite par un chemin clos (*clipping*).

Les autres types METAPOST (chaînes de caractères, booléens, couleurs) sont directement représentés en OCaml. L'interface de MLPOST contient aussi un module `Command` qui définit le type des commandes METAPOST : commandes de dessin, de remplissage, itérations, séquences, etc.

Dépendances circulaires. La réalisation de ces modules de bas niveau présente quelques difficultés. Premièrement, la plupart des modules présentés sont *a priori* mutuellement récursifs : par exemple, les transformations s'appliquent à tous les autres objets, donc chaque module contient une fonction

```
val transform : Transform.t → t → t
```

où le type `t` représente le type principal du module en question. D'un autre côté, les transformations sont elles-mêmes construites à l'aide de numériques et de points :

```
val shifted : Point.t → t
```

où `t` est le type des transformations. Des dépendances circulaires existent aussi entre types et sont aggravées par la représentation symbolique des objets (par exemple, les projections `xpart` et `ypart` du module `Point` doivent retourner des numériques et non des flottants).

Nous souhaitons réaliser ces différents modules dans des fichiers différents mais OCaml ne permet pas de dépendances circulaires entre des fichiers. Notre solution consiste à définir tous les *types* dans un seul fichier `types.mli`. Chaque module fait maintenant référence à ce fichier. Par exemple, dans le fichier `path.ml`, qui fournit l'implémentation du module `Path`, on trouvera

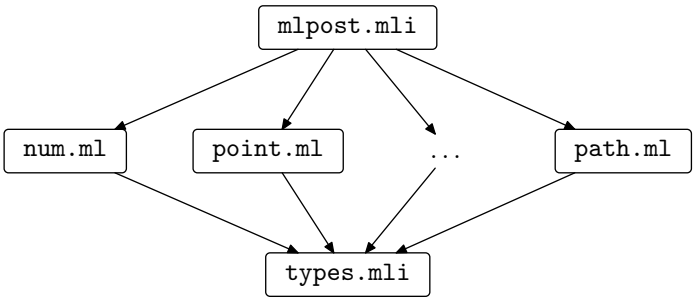
```
type t = Types.path
```

La dépendance circulaire entre les modules est ainsi cassée de manière très classique. En revanche, on souhaite cacher l'existence du module `Types` pour les deux raisons suivantes :

- la clarté des messages d'erreur ;
- la clarté de la documentation générée par `ocamldoc`.

On souhaite donc rétablir la circularité entre les modules au sein de l'interface du module `Mlpost`. Pour cela, on écrit un unique fichier `mlpost.mli` qui contient les définitions des signatures des modules à exporter :

```
module rec Num : sig
  type t
  ...
end
and Point : sig
  type t
  val xpart : t → Num.t
  ...
end
and Path : sig
  type t
  ...
end
...
```

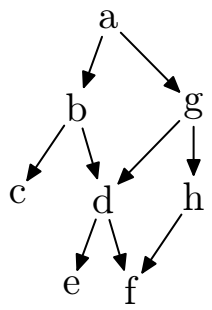


Toutes ces signatures sont déclarées de manière mutuellement récursive. La signature de `Point` peut ainsi faire référence à `Path` et inversement. On notera que les types tels que `Point.t` sont maintenant abstraits dans l'interface, rendant invisible l'existence du module `Types`. Par ailleurs, les implémentations de ces modules sont contenues dans des fichiers indépendants `num.ml`, `point.ml`, etc., qui sont compilés avec l'option `-for-pack Mlpost`. Cet agencement est en outre très pratique pour la documentation de l'API : c'est uniquement le fichier `mlpost.mli` qui sert d'entrée à l'outil `ocamldoc`.

Hash-consing et traduction vers METAPOST. Le choix d'une représentation symbolique de la plupart des objets impose des efforts supplémentaires pour minimiser l'utilisation de la mémoire et la taille des fichiers METAPOST de sortie. En effet, l'arbre de syntaxe abstraite (AST) contient beaucoup de nœuds identiques mais construits de manière différente, qui prennent donc inutilement de la place aussi bien en mémoire que dans le fichier METAPOST généré. C'est d'autant plus gênant que METAPOST devra lire ce fichier et passera donc davantage de temps sur des calculs répétés.

Pour y remédier, nous utilisons la technique du *hash-consing* [6] appliquée à l'arbre de syntaxe abstraite. Cette technique permet de partager des valeurs structurellement égales. Elle utilise une table de hachage globale qui stocke toutes les valeurs déjà créées. Avant de créer un nouvel objet, on regarde dans cette table si un objet structurellement égal existe déjà. Pour que le calcul de la valeur de hachage soit efficace, chaque (sous-)terme vient avec sa valeur de hachage. Le partage réalisé est maximal, ce qui permet de substituer l'égalité physique (`==`) à l'égalité structurelle (`=`).

Cette technique diminue l'utilisation de la mémoire, mais ne change rien *a priori* à la taille des fichiers générés. La structure hash-consée *réalise* le partage, mais elle ne sait pas quels sont les nœuds effectivement utilisés au moins deux fois. Pour cela, on réalise un simple parcours en profondeur de la structure, en comptant les occurrences de chaque nœud. Il est néanmoins possible de rencontrer une nouvelle fois un sous-nœud d'une structure, comme le montre l'exemple ci-dessous :



Dans cette configuration, le nœud f est réellement utilisé deux fois, alors que le nœud e n'est utilisé qu'une seule fois, par le nœud d, même si celui-ci est utilisé deux fois à son tour. Autrement dit, on comptabilise pour chaque nœud le nombre de flèches incidentes.

Après cette analyse, la génération du fichier METAPOST devient très simple : il suffit de traverser de nouveau l'arbre de syntaxe abstraite et, quand on visite un nœud qui est utilisé au moins deux fois, on construit une définition METAPOST pour cet objet. Il faut néanmoins prendre en compte les particularités syntaxiques de METAPOST telles que la précédence inhabituelle des opérateurs arithmétiques et la restriction de l'application de certaines constructions à des variables. De cette façon, on arrive à avoir du code METAPOST relativement petit [5], malgré la délégation des calculs à METAPOST.

5. En l'absence de boucles `for` et de macros dans le code METAPOST, nous avons observé, sans avoir fait de tests très exhaustifs, une taille du code généré du même ordre de grandeur que celle du code METAPOST écrit à la main.

3.2. Boîtes

Comme nous l'avons illustré déjà maintes fois, les boîtes de MLPOST peuvent être réduites à de simples objets LaTeX ou bien être constituées d'un ensemble d'autres boîtes. Le type des boîtes est donc un type récursif de la forme suivante :

```
type t =
  { name : string option;
    width : Num.t;  height : Num.t; pen : Pen.t option; ...
    desc : desc; }

and desc =
  | Emp
  | Pic of Picture.t
  | Grp of t array × t Smap.t
```

Chaque boîte est éventuellement nommée (champ `name`), possède un certain nombre d'attributs (position, taille, couleur, bordure, remplissage, etc.) et sa nature est donnée par le champ `desc`. Ce dernier indique s'il s'agit d'une boîte vide, d'une image ou bien d'une boîte composite. Dans ce dernier cas, les sous-boîtes sont contenues dans un tableau, qui est accompagné d'une table (réalisée par le module `Smap`) permettant un accès plus rapide à une sous-boîte par son nom.

Le dessin d'une boîte est immédiat. On trace d'une part son contour et d'autre part son contenu. Ce dernier est soit une boîte atomique directement dessinée à l'aide de `Command.draw_pic`, soit une boîte composite dont le dessin est tout simplement obtenu en dessinant récursivement chaque sous-boîte.

Pour réaliser les diverses fonctions de placement, on commence par écrire une fonction de translation d'une boîte par un vecteur donné :

```
Box.shift : Point.t → Box.t → Box.t
```

Cette fonction est naturellement récursive sur la structure de la boîte. Il est important de noter que cette fonction renvoie une *nouvelle* boîte, sans altérer son argument (les boîtes sont persistantes). Une fois cette fonction donnée, il est aisé de réaliser les fonctions `Box.hbox`, `Box.hblock`, etc.

3.3. Flèches

METAPOST ne propose qu'un seul type de flèche. Une flèche METAPOST suit un chemin arbitraire mais son tracé est limité aux différents styles de trait (plume et pointillés) et la tête de flèche est toujours la même :

Avec MLPOST, l'utilisateur peut créer ses propres catégories de flèches à l'aide du module `Arrow`. Celui-ci propose deux types :

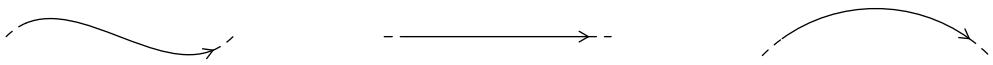
- Le type `head` décrit comment dessiner une tête de flèche. Les éléments de type `head` sont des fonctions prenant en argument la position et la direction de la tête de flèche et renvoyant une commande dessinant la tête de flèche.
- Le type abstrait `kind` décrit une catégorie de flèche. Une catégorie décrit les différents éléments dans le dessin d'une flèche, les têtes de flèche pouvant en réalité être placées n'importe où le long de la flèche. Pour construire une nouvelle catégorie, on part de la catégorie vide et on ajoute des traits et des têtes.

La fonction `draw` permet de dessiner une flèche d'une catégorie donnée en suivant un chemin donné. Les flèches sont alors dessinées en utilisant les primitives de METAPOST. Ceci a l'inconvénient d'utiliser plus de ressources mais permet d'imaginer de nombreuses catégories de flèches.

On peut en particulier retrouver les flèches de METAPOST. On part d'un corps vide et on lui ajoute un trait normal sur toute la longueur. On ajoute enfin une tête triangulaire remplie, et on obtient :


```
let kind =
  Arrow.add_head ~head:Arrow.head_triangle_full
    (Arrow.add_line Arrow.empty) in
[ Arrow.draw ~kind (...path...) ]
```

La section 2.2 contient une figure décrivant le fonctionnement de la fonction `loop`. Pour les besoins de cette figure, on a créé un type de flèche spécial, composé d'un début et d'une fin en pointillés et avec une tête placée différemment :



Le code permettant d'obtenir cette catégorie de flèche est le suivant :

```
let kind =
  Arrow.add_belt ~point:0.9
    (Arrow.add_line ~dashed:Dash.evenly ~to_point:0.1
       (Arrow.add_line ~dashed:Dash.evenly ~from_point:0.9
          (Arrow.add_line ~from_point:0.1 ~to_point:0.9
             Arrow.empty)))
```

4. Conclusion

Nous avons présenté MLPOST, une bibliothèque OCaml au dessus de METAPOST. Nous espérons avoir convaincu le lecteur des avantages que la présentation sous forme de bibliothèque apporte : familiarité avec le langage pour les programmeurs OCaml, typage fort, des constructions de programmation de haut niveau, des dessins résultants de calculs arbitraires, etc. MLPOST fournit volontairement un nombre restreint de primitives, car l'utilisateur peut aisément construire des extensions au dessus de MLPOST. En cela, MLPOST diffère de bibliothèques LaTeX telles que TikZ ou PSTricks, où de très nombreuses fonctionnalités sont fournies mais où il est très difficile d'en ajouter pour qui ne maîtrise pas TeX.

L'une des forces de MLPOST est de proposer un style déclaratif, là où la majorité des bibliothèques graphiques propose un style impératif. Ceci permet en particulier un *partage* immédiat de sous-éléments dans une ou plusieurs figures. Une autre force de MLPOST est le typage statique directement hérité d'OCaml. On évite ainsi l'immense majorité des erreurs à l'exécution de METAPOST, souvent cryptiques. Il reste néanmoins les erreurs éventuellement contenues dans les extraits de LaTeX ou les erreurs de nature géométrique telles que le remplissage d'un chemin en forme de 8. Nous pourrions envisager d'utiliser des types OCaml plus précis, par exemple pour distinguer les chemins clos et non clos ou encore les points et les vecteurs.

Il reste une fonctionnalité intéressante de METAPOST qui n'est pas interfacée dans MLPOST : la résolution d'équations linéaires. Il y a deux raisons à cela. D'une part, les équations servent souvent au placement implicite, et MLPOST fournit une alternative sous la forme de fonctions d'alignement de boîtes. D'autre part, la résolution d'équations de METAPOST procède de manière impérative et il n'est pas simple de l'intégrer dans le contexte déclaratif qui est le nôtre. Ceci étant dit, il serait intéressant d'explorer des méthodes de placement plus automatiques que celles que nous proposons, par exemple inspirées de la manière dont TeX mets en page lignes, pages et paragraphes.

Enfin, il est important de noter que MLPOST n'est pas lié à METAPOST de manière intrinsèque. On pourrait facilement ajouter une sortie TikZ, ou même directement une sortie PostScript à condition d'utiliser une technique similaire à celle de METAPOST pour l'inclusion de LaTeX.

Remerciements

Les auteurs tiennent à remercier Florence Plateau, Yannick Moy et Claude Marché pour leur contribution à MLPOST, Sylvie Boldo pour la suggestion du titre de l'article et les relecteurs pour leurs remarques.

Références

[1] Le langage Haskell. `http://www.haskell.org/`.

[2] Le langage Objective Caml. `http://caml.inria.fr/`.

[3] Jack E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1) :25–30, January 1965.

[4] Emmanuel Chailloux and Ascánder Suárez. mlPicTeX, a picture environment for LaTeX. In *Workshop on ML*, pages 79–90, 1994.

[5] Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. In *The Sixth ASIAN Symposium on Programming Languages and Systems (APLAS)*, Bangalore, India, December 2008.

[6] Sylvain Conchon and Jean-Christophe Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

[7] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1) :86–124, 1989.

[8] T. Van Zandt et al. PSTricks. `http://tug.org/PSTricks/`.

[9] Meik Hellmund, Ralf Hinze, Joachim Korittky, Marco Kuhlmann, Ferenc Wágner, and Peter Simons. *functional* METAPOST. `http://cryp.to/funcmp/`.

[10] John Hobby. METAPOST, 1994. `http://plan9.bell-labs.com/who/hobby/MetaPost.html`.

[11] Donald E. Knuth. *The METAFONT Book*. Addison-Wesley, 1984.

[12] Alexander Larsson. Dia. `http://live.gnome.org/Dia`.

[13] Glenn C. Reid. *PostScript Language Program Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[14] Brian V. Smith. Xfig. `http://www.xfig.org/`.

[15] Till Tantau. PGF and TikZ – Graphic systems for TeX. `http://sourceforge.net/projects/pgf/`.