Actes des vingt-cinquièmes journées francophones des langages applicatifs

(JFLA)

Fréjus, du 8 au 11 janvier 2014

Cet ouvrage réunit les contributions qui ont été présentées lors des vingt-cinquièmes journées francophones des langages applicatifs. Les douze articles choisis par le comité de programme reflètent la diversité de notre communauté, en couvrant des domaines allant de la programmation fonctionnelle à la preuve formelle, en passant par les modèles théoriques sous-jacents.

Nous remercions chaleureusement tous les membres du comité de programme et les relecteurs additionnels pour leur aide précieuse lors du processus de sélection, ainsi que toutes les personnes ayant soumis un article ou participé à ces journées. Merci également à Olivier Danvy et à Christine Paulin d'avoir accepté de venir donner un cours lors des journées, ainsi qu'à Guillaume Brunerie, Jean Krivine, Catherine Lelay, et Xavier Leroy, d'avoir accepté l'invitation à venir faire un exposé.

Ces journées n'auraient pas pu avoir lieu sans l'aide de Sophie Azzaro, du bureau des cours et colloques de l'Inria Grenoble Rhône-Alpes ; nous lui en sommes extrêmement reconnaissants. Nous remercions aussi Hélène Lowinger, du service Information Scientifique et Communication de l'Inria Saclay - Île-de-France, qui a assuré le dépot de ces actes sous la forme d'une collection HAL.

Christine Tasson, Université Paris Diderot – PPS

David Baelde, ÉNS Cachan – LSV

Comité de programme

Jade Alglave	University College of London
David Baelde	LSV – ÉNS Cachan (Vice président)
Zaynah Dargaye	CEA LIST
Jean-Christophe Filliâtre	CNRS – Université Paris Sud
Pascal Fradet	INRIA Grenoble – Rhône-Alpes
Jacques Garrigue	Nagoya University
Barbara Petit	INRIA Grenoble – Rhône Alpes
Sylvain Pogodolla	Loria – INRIA Nancy
Sylvain Pradalier	Dassault Systèmes
Julien Signoles	CEA LIST
Matthieu Sozeau	PPS – INRIA
Christine Tasson	PPS – Université Paris VII (Présidente)

Relecteurs additionnels

Mounir Assaf	CEA LIST
Vincent Balat	PPS – Université Paris VII
Thomas Braibant	INRIA Rocquencourt
Philippe De Groote	INRIA Nancy – Grand Est
Stéphane Gimenez	University of Innsbruck
Antoine Madet	ÉNS – INRIA
Damiano Mazza	LIPN – Université Paris XIII
Samuel Mimram	CEA LIST
Andrei Paskevich	LRI – Université Paris Sud
Matthias Puech	Aarhus University
Kazushige Terui	RIMS – Kyoto University

Table des matières

Articles

Communications courtes

Formal verification in Coq of program properties involving the global state effect

Jean-Guillaume Dumas[1], Dominique Duval[1], Burak Ekici[1] & Damien Pous[2]

1: LJK, Université de Grenoble, France. Firstname.Lastname@imag.fr
2: LIP, ENS Lyon, France. Damien.Pous@ens-lyon.fr

Abstract

The syntax of an imperative language does not mention the state explicitly, while its denotational semantics (*interpretations*) has to mention it. This paper presents a Coq framework for verifying the properties of the programs with state manipulation. These properties are expressed in a proof system which is close to the syntax, as in effect systems, in the sense that the state does not appear explicitly in the type of expressions which manipulate it. Rather, the state appears via decorations (*annotations*) added to terms and to equations. In this system, proofs of programs thus present two aspects: properties can be verified *up to effects* or the effects can be taken into account. The design of our Coq library consequently reflects these two aspects: our framework is centered around the construction of two inductive and dependent types, one for terms up to effects and one for the manipulation of decorations.

1. Introduction

The evolution of the memory state in an imperative program is a computational effect. For instance, a `C` function updating the value of a variable has an effect on the state. However, the state is never mentioned as an argument or a result of a command, whereas in general it is used and modified during the execution of commands. Thus, the syntax of an imperative language does not mention the state explicitly, while its denotational semantics (in which the meanings of syntactic expressions are interpreted by mathematical objects) has to mention it. This means that the state is encapsulated: its interface, which is made of the functions for looking up and updating the values of the locations, is separated from its implementation; the state cannot be accessed in any other way than through its interface. These two points of view (encapsulating or expliciting the state) are also valid for proofs. For instance, the fact that modifying the value of a variable and observing the value of another variable can be done in any order is called the *update-lookup commutation*. This property may be proved by expliciting the state, typically as an array. However, it turns out that it may also be proved without any knowledge of the implementation of the state. This is made possible by adding decorations (or annotations) to terms, as in effect systems [8, 13], and also to equations, as in [3]. The decoration of a term (or an equation) characterizes the way it may interact with the state; for instance there is a decoration for terms which are *read-only* and another one for terms which are *read-write*. This novel approach enables a verification of proofs in two steps: a first step checks the syntax *up to effects* by dropping the decorations; a second step then takes the effects into account. This approach uses categorical constructions to model the denotational semantics of the state effect and prove some properties of programs involving this effect. *Strong monads*, introduced by Moggi [9], were the first categorical approach to computational effects, while Power et al. [11] then proposed the *premonoidal categories*. Next Hughes [7] extended Haskell with *arrows* that share some properties with the approach of *cartesian effect categories* of Dumas et al. [5].

The main goal of this paper is to present a framework in the Coq proof assistant which enables programmers to verify properties of programs involving the state effect. Proving properties of programs involving the state

effect is important when the order of evaluation of the arguments is not specified or more generally when parallelization comes into play [8].

The salient feature of our framework is that it is possible to manipulate these properties and their proofs using decorations for the state effect, and thus to remain close to syntax. Then we show that it is possible to verify in Coq the well known properties of the state effect described, e.g., in [10]. More precisely, it is now possible to check proofs of properties of imperative programs without knowing the exact implementation of the state. As far as we know, such a tool does not exist yet; in this first version we focus on composition of lookups and updates.

To some extent, our work is quite similar to [2] in the sense that we also define our own programming language and verify its properties by using axiomatic semantics. We construct our system on categorical notions (e.g. monads) as done in [1]. In brief, we first declare our system components including their properties and then prove some related propositions. In that manner, the overall idea is also quite close to [12], even though technical details completely differ. However, to our knowledge, the use of decorations in Coq for hiding the implementation details is new.

In this paper, we show that the decorated proof system can be developed in Coq thus enabling a mechanised verification of decorated proofs for side-effect systems. We recall in Section 2 the logical environment for decorated equational proofs involving the state effect. Then in Section 3 we present the translation of the categorical rules into Coq as well as their resulting derivations and the necessary additions. The resulting Coq code has been integrated into a library, available there: `http://coqeffects.forge.imag.fr`. Finally, in Section 4 we give the full details of the proof of the update-lookup commutation property and its verification in Coq, as an example of the capabilities of our library.

2. The Logical Environment for Equational Proofs

2.1. Motivation

Essentially, in a purely functional programming language, an operation or a term f with an argument of type X and a result of type Y, which may be written $f : X \to Y$ (in the *syntax*), is interpreted (in the *denotational semantics*) as a function $[\![f]\!]$ between the sets $[\![X]\!]$ and $[\![Y]\!]$, interpretations of X and Y. It follows that, when an operation has several arguments, these arguments can be evaluated in parallel, or in any order. It is possible to interpret a purely functional programming language via a categorical semantics based on *cartesian closed categories*; the word "cartesian" here refers to the categorical *products*, which are interpreted as *cartesian products* of sets, and which are used for dealing with pairs (or tuples) of arguments. The *logical semantics* of the language defines a set of rules that may be used for proving properties of programs.

But non-functional programming languages such as `C` or `Java` do include computational effects. For instance a `C` function may modify the state structure and a `Java` function may throw an exception during the computation. Such operations are examples of computational effects. In this paper we focus on the state effect. We consider the *lookup* and *update* operations for modeling the behavior of imperative programs: namely an *update* operation assigns a value to a location (or variable) and a *lookup* operation recovers the value of a location. There are many ways to handle computational effects in programming languages. Here we focus on the categorical treatment of [5], adapted to the state effect [3]: this provides a logical semantics relying on *decorations*, or annotations, of terms and equations.

2.2. Decorated functions and equations for the state effect

The functions in our language are classified according to the way they interact with the state. The classification takes the form of annotations, or decorations, written as superscripts. A function can be a *modifier*, an *accessor* or a *pure* function.

- As the name suggests, a *modifier* may modify or use the state: it is a *read-write* function. We will use the keyword *rw* as an annotation for modifiers.
- An *accessor* may use the state structure but never modifies it: it is a *read-only* function. We will use the keyword *ro* for accessors.
- A *pure function* never interacts with the state. We will use the keyword *pure* for pure functions.

The denotational semantics of this language is given in terms of the set of states S and the *cartesian product* operator '$\times$'. For all types X and Y, interpreted as sets $[\![X]\!]$ and $[\![Y]\!]$, a modifier function $f : X \to Y$ is interpreted as a function $[\![f]\!] : [\![X]\!] \times S \to [\![Y]\!] \times S$ (it can access the state and modify it); an accessor g as $[\![g]\!] : [\![X]\!] \times S \to [\![Y]\!]$ (it can access the state but not modify it); and a pure function h as $[\![h]\!] : [\![X]\!] \to [\![Y]\!]$ (it can neither access nor modify the state). There is a hierarchy among those functions. Indeed any pure function can be seen as both an accessor or a modifier even though it will not actually use its argument S. Similarly an accessor can be seen as a modifier.

The state is made of memory *locations*, or *variables*; each location has a value which can be updated. For each location i, V_i is the type of the values that can be stored in the location i, and $Val_i = [\![V_i]\!]$ is its interpretation. The unit type (denoted, e.g., `void` in some programming languages) is here denoted by $\mathbb{1}$; its interpretation is a singleton, also denoted by $\mathbb{1}$. The assignment of a value of type V_i to a variable i takes an argument of type V_i. It does not return any result but it modifies the state: given a value $a \in Val_i$, the assignment of a to i sets the value of location i to a and keeps the value of the other locations unchanged. Thus, this operation is a modifier from V_i to $\mathbb{1}$. It is denoted by $update_i^{rw} : V_i \to \mathbb{1}$ and it is interpreted as $[\![update_i]\!] : Val_i \times S \to S$. The recovery of the value stored in a location i takes no argument and returns a value of type V_i. It does not modify the state but it observes the value stored at location i. Thus, this operation is an accessor from $\mathbb{1}$ to V_i. It is denoted by $lookup_i^{ro} : \mathbb{1} \to V_i$ and it is interpreted (since $\mathbb{1} \times S$ is in bijection with S) as $[\![lookup_i]\!] : S \to Val_i$. For each type X, the *identity* operation $id_X : X \to X$, which is interpreted by mapping each element of $[\![X]\!]$ to itself, is pure. Similarly, the *final* operation $\langle\rangle_X : X \to \mathbb{1}$, is interpreted by a mapping each element of $[\![X]\!]$ to the unique element of $\mathbb{1}$, is pure. To simplify, we will often use id_i and $\langle\rangle_i$ instead of respectively id_{Val_i} and $\langle\rangle_{Val_i}$.

In addition, decorations are also added to equations. In that parallel, two functions $f, g : X \to Y$ are said to be *strongly equal* if they return the same result and have the same effect on the state structure. This is denoted $f == g$. They are called *weakly equal* if they return the same result but may have different effects on the state. This is denoted $f \sim g$.

The state can be observed thanks to the lookup functions. For each location i, the interpretation of the $update_i$ operation is characterized by the following equalities, for each state $s \in S$ and each $x \in Val_i$:

$$\begin{cases} [\![lookup_i]\!]([\![update_i]\!](s,x)) = x \\ [\![lookup_j]\!]([\![update_i]\!](s,x)) = [\![lookup_j]\!](s) \text{ for every } j \in Loc,\ j \neq i \end{cases}$$

According to the previous definitions, these equalities are the interpretations of the following weak equations:

$$\begin{cases} lookup_i^{ro} \circ update_i^{rw} \sim id_i^{pure} : V_i \to V_i \\ lookup_j^{ro} \circ update_i^{rw} \sim lookup_j^{ro} \circ \langle\rangle_i^{pure} \text{ for every } j \in Loc,\ j \neq i : V_i \to V_j \end{cases}$$

For instance, suppose that two given variables i and j are of integer type $[\![V_i]\!] = [\![V_j]\!] =$ `int`. Then the C program `i=3;j=i;` can be modeled as the command $update_j^{rw} \circ lookup_i^{ro} \circ update_i^{rw} \circ 3^{pure} : \mathbb{1} \to \mathbb{1}$, where constants are modeled as pure functions from $\mathbb{1}$, as is $3^{pure} : \mathbb{1} \to V_i$.

2.3. Sequential Evaluation of Arguments

To represent pure functions taking multiple arguments in a categorical way, one usually uses cartesian *products*. For instance, two functions $[\![f_1]\!] : [\![X_1]\!] \to [\![Y_1]\!]$ and $[\![f_2]\!] : [\![X_2]\!] \to [\![Y_2]\!]$ give rise to a third function $[\![f_1 \times f_2]\!] : [\![X_1]\!] \times [\![X_2]\!] \to [\![Y_1]\!] \times [\![Y_2]\!]$. Similarly the action of a function $[\![f]\!] : [\![X]\!] \to [\![Y_1]\!] \times [\![Y_2]\!]$, returning a *pair*

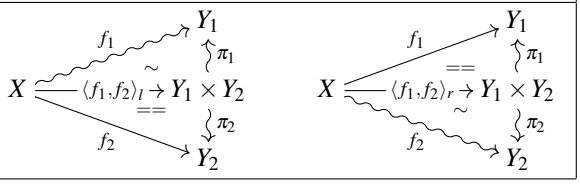
of results can be classically seen as the resulting action of two functions, $f_1 : [\![X]\!] \to [\![Y_1]\!]$ and $f_2 : [\![X]\!] \to [\![Y_2]\!]$. We will recall next how to model this, directly in the decorated model, using projections and their interpretations from $Y_1 \times Y_2$ to Y_1 and Y_2.

While the order of evaluation in such pairs or products of *pure* functions does not matter, when considering functions with side-effects (typically, updates of the state), one has to specify the evaluation order: the results can depend on this choice. To this end, we use *sequential products* of functions, as introduced in [5], which impose some order of evaluation of the arguments: $f_1 \ltimes f_2$ denotes *left sequential products* where f_1 is evaluated before f_2. Symmetrically $f_1 \rtimes f_2$ denotes *left sequential products* where f_2 is evaluated before f_1. A sequential product is obtained as the sequential composition of two *semi-pure products*. A semi-pure product, as far as we are concerned in this paper, is a kind of product of an identity function (which is pure) with another function (which may be any modifier).

For each types X and Y, we introduce a *product* type $X \times Y$ with *projections* $\pi^{pure}_{1,X_1,X_2} : X_1 \times X_2 \to X_1$ and $\pi^{pure}_{2,X_1,X_2} : X_1 \times X_2 \to X_2$, which will be denoted simply by π^{pure}_1 and π^{pure}_2. This is interpreted as the cartesian product with its projections. Pairs and products of pure functions are built as usual. In the special case of a product with the unit type, it can easily be proved, as usual, that $\pi^{pure}_1 : X \times \mathbb{1} \to X$ is invertible and the inverse is stated as follows: $(\pi_1^{-1})^{pure} = \langle id^{pure}_X, \langle\rangle^{pure}_X \rangle : X \to X \times \mathbb{1}$. The *permutation* operation $perm_{X\times Y} : X \times Y \to Y \times X$ is also pure: it is interpreted as the function which exchanges its two arguments.

Given a pure function $f_1^{pure} : X \to Y_1$, interpreted as $[\![f_1]\!] : [\![X]\!] \to [\![Y_1]\!]$, and a modifier $f_2^{rw} : X \to Y_2$ with its interpretation $[\![f_2]\!] : [\![X]\!] \times S \to [\![Y_2]\!] \times S$, the *left semi-pure pair* $\langle f_1, f_2 \rangle^{rw}_l : X \to Y_1 \times Y_2$ is the modifier interpreted by $[\![\langle f_1, f_2\rangle_l]\!] : [\![X]\!] \times S \to [\![Y_1]\!] \times [\![Y_2]\!] \times S$ such that $[\![\langle f_1, f_2\rangle_l]\!](x,s) = (y_1, y_2, s')$ where $y_1 = [\![f_1]\!](x)$ and $(y_2, s') = [\![f_2]\!](x,s)$. The left semi-pure pair $\langle f_1, f_2\rangle^{rw}_l$ is characterized, up to strong equations, by a weak and a strong equation: $\pi^{pure}_1 \circ \langle f_1, f_2\rangle^{rw}_l \sim f_1^{pure}$ and $\pi^{pure}_2 \circ \langle f_1, f_2\rangle^{rw}_l == f_2^{rw}$. Indeed, this pair has the same result on Y_1 (resp. Y_2) as f_1 (resp. f_2) alone. Moreover, it as the same effect as f_2 alone, but not as f_1, since the latter is pure. The *right semi-pure pair* $\langle f_1, f_2\rangle^{rw}_r : X \to Y_1 \times Y_2$ where $f_1^{rw} : X \to Y_1$ and $f_2^{pure} : X \to Y_2$ is defined in the symmetric way[1]:

Left & Right Semi-Pure Pairs



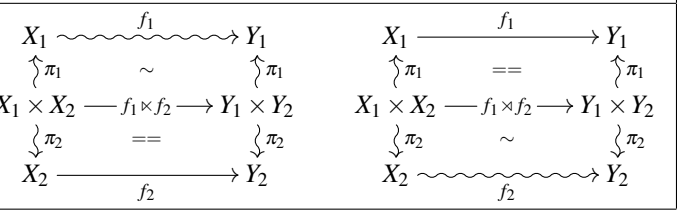
(1)

The *left semi-pure product* is defined in the usual way from the left semi-pure pair: given $f_1^{pure} : X_1 \to Y_1$ $f_2^{rw} : X_2 \to Y_2$, the left semi-pure product of f_1 and f_2 is $(f_1 \ltimes f_2)^{rw} = \langle f_1 \circ \pi_{1,X_1,X_2}, f_2 \circ \pi_{2,X_1,X_2} \rangle^{rw}_l : X_1 \times X_2 \to Y_1 \times Y_2$. It is characterized, up to strong equations, by a weak and a strong equation:

$$\pi^{pure}_{1,Y_1,Y_2} \circ (f_1 \ltimes f_2)^{rw} \sim f_1^{pure} \circ \pi^{pure}_{1,X_1,X_2} \text{ and } \pi^{pure}_{2,Y_1,Y_2} \circ (f_1 \ltimes f_2)^{rw} == f_2^{rw} \circ \pi^{pure}_{2,X_1,X_2} \tag{2}$$

The *right semi-pure product* $(f_1 \rtimes f_2)^{rw} : X_1 \times X_2 \to Y_1 \times Y_2$ is defined in the symmetric way:

Left & Right Semi-Pure Products



(3)

[1] In all diagrams, the decorations are expressed by shapes of arrows: waving arrows for pure functions, dotted arrows for accessors and straight arrows for modifiers.

Now, it is easy to define the *left sequential product* of two modifiers $f_1^{rw} : X_1 \to Y_1$ and $f_2^{rw} : X_2 \to Y_2$ by composing a right semi-pure product with a left semi-pure one and using *id* function as the pure component:

$$(f_1 \ltimes f_2)^{rw} = (id_{Y_1} \ltimes f_2)^{rw} \circ (f_1 \rtimes id_{X_2})^{rw} \colon X_1 \times X_2 \to Y_1 \times Y_2$$

In a symmetric way, the *right sequential product* of $f_1^{rw} : X_1 \to Y_1$ and $f_2^{rw} : X_2 \to Y_2$ is defined as:

$$(f_1 \rtimes f_2)^{rw} = (f_1 \rtimes id_{Y_2})^{rw} \circ (id_{X_1} \ltimes f_2)^{rw} \colon X_1 \times X_2 \to Y_1 \times Y_2$$

The left sequential product models the fact of executing f_1 before f_2, while the right sequential product models the fact of executing f_2 before f_1; in general they return different results and they modify the state in a different way.

Left & Right Sequential Products

$$\begin{array}{ccccc} X_1 & \xrightarrow{f_1} & Y_1 & \overset{id}{\rightsquigarrow} & Y_1 \\ \uparrow\pi_1 & == & \uparrow\pi_1 & \sim & \uparrow\pi_1 \\ X_1 \times X_2 & \xrightarrow{f_1 \rtimes id} & Y_1 \times X_2 & \xrightarrow{id \ltimes f_2} & Y_1 \times Y_2 \\ \downarrow\pi_2 & \sim & \downarrow\pi_2 & == & \downarrow\pi_2 \\ X_2 & \overset{id}{\rightsquigarrow} & X_2 & \xrightarrow{f_2} & Y_2 \end{array} \qquad \begin{array}{ccccc} X_1 & \overset{id}{\rightsquigarrow} & X_1 & \xrightarrow{f_1} & Y_1 \\ \uparrow\pi_1 & \sim & \uparrow\pi_1 & == & \uparrow\pi_1 \\ X_1 \times X_2 & \xrightarrow{id \ltimes f_2} & Y_1 \times X_2 & \xrightarrow{f_1 \rtimes id} & Y_1 \times Y_2 \\ \downarrow\pi_2 & == & \downarrow\pi_2 & \sim & \downarrow\pi_2 \\ X_2 & \xrightarrow{f_2} & Y_2 & \overset{id}{\rightsquigarrow} & Y_2 \end{array} \tag{4}$$

2.4. A property of the state

In [10] an equational presentation of the state is given, with seven equations. These equations explain the behaviors of the main prototype operations taking place on the global state structure. They are the archetype of the properties of the proofs we want to verify. For instance, the fact that modifying a location i and observing the value of another location j can be done in any order is called the *commutation update-lookup* property. This property can be expressed as an equation relating the functions $[\![update_i]\!]$ and $[\![lookup_j]\!]$. For this purpose, let $[\![lookup_j]\!]' : S \to Val_j \times S$ be defined by

$$[\![lookup_j]\!]'(s) = ([\![lookup_j]\!](s), s) \text{ for each } s \in S\,.$$

Thus, given a state s and a value $a \in Val_i$, assigning a to i and then observing the value of j is performed by the function:

$$[\![lookup_j]\!]' \circ [\![update_i]\!] \colon Val_i \times S \to Val_j \times S\,.$$

Observing the value of j and then assigning a to i also corresponds to a function from $Val_i \times S$ to $Val_j \times S$ built from $[\![update_i]\!]$ and $[\![lookup_j]\!]'$. This function first performs $[\![lookup_j]\!]'(s)$ while keeping a unchanged, then it performs $[\![update_i]\!](s,a)$ while keeping b unchanged (where b denotes the value of j in s which has been returned by $[\![lookup_j]\!](s)$). The first step is $id_{Val_i} \times [\![lookup_j]\!]' \colon Val_i \times S \to Val_i \times (Val_j \times S)$ and the second step is $id_{Val_j} \times [\![update_i]\!] \colon Val_j \times (Val_i \times S) \to Val_j \times S$. An intermediate permutation step is required, it is called $perm_{i,j} \colon Val_i \times (Val_j \times S) \to Val_j \times (Val_i \times S)$ such that $perm_{i,j}(a,(b,s)) = (b,(a,s))$. Altogether, observing the value of j and then assigning a to i corresponds to the function:

$$(id_{Val_j} \times [\![update_i]\!]) \circ perm_{i,j} \circ (id_{Val_i} \times [\![lookup_j]\!]') \colon Val_i \times S \to Val_j \times S$$

Thus, the commutation update-lookup property means that:

$$[\![lookup_j]\!]' \circ [\![update_i]\!] = (id_{Val_j} \times [\![update_i]\!]) \circ perm_{i,j} \circ (id_{Val_i} \times [\![lookup_j]\!]')$$

According to Section 2.2, this is the interpretation of the following strong equation, which can also be expressed as a diagram:

$$lookup_j^{ro} \circ update_i^{rw} == \pi_2^{pure} \circ (update_i^{rw} \rtimes id_j^{pure}) \circ (id_i^{pure} \ltimes lookup_j^{ro}) \circ (\pi_1^{-1})^{pure} : V_i \to V_j\,. \tag{5}$$

Commutation: Update & Lookup

$$V_i \xrightarrow{update_i} \mathbb{1} \overset{lookup_j}{\cdots\!\rightarrow} V_j \quad == \quad V_i \overset{\pi_1^{-1}}{\rightsquigarrow} V_i \times \mathbb{1} \overset{id_i \ltimes lookup_j}{\cdots\!\rightarrow} V_i \times V_j \xrightarrow{update_i \rtimes id_j} \mathbb{1} \times V_j \overset{\pi_2}{\rightsquigarrow} V_j$$

Remark. Using the right sequential product, the right hand-side of the commutation update-lookup equation can be written as $\pi_2^{pure} \circ (update_i^{rw} \rtimes lookup_j^{ro}) \circ (\pi_1^{-1})^{pure}$. In addition, using the left sequential product, it is easy to check that the left hand-side of this equation can be written as $\pi_2^{pure} \circ (update_i^{rw} \ltimes lookup_j^{ro}) \circ (\pi_1^{-1})^{pure}$. Since $\pi_1^{pure} : V_i \times \mathbb{1} \to V_i$ and $\pi_2^{pure} : \mathbb{1} \times V_j \to V_j$ are invertible, we get a symmetric expression for the equation which corresponds nicely to the description of the commutation update-lookup property as "the fact that modifying a location i and observing the value of another location j can be done in any order":

$$update_i^{rw} \rtimes lookup_j^{ro} == update_i^{rw} \ltimes lookup_j^{ro}$$

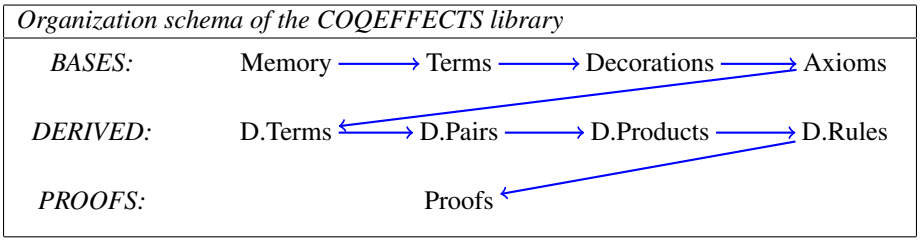
3. The Environment in Coq

In this section we present the core of this paper, namely the implementation in the Coq proof assistant of the rules for reasoning with decorated operations and equations and the proof of the update-lookup commutation property using these rules.

In Section 4, we have shown proofs of propositions involving effects. We now present the construction of a Coq framework enabling one to formalize such proofs. This framework has been released as `STATES-0.5` library and is available in the following web-site: `http://coqeffects.forge.imag.fr`.

In order to construct this framework, we need to define data structures, terms, decorations and basic rules as axioms. Those give rise to derived rules and finally to proofs. This organization is reflected in the library with corresponding Coq modules, as shown in the following diagram:

Organization schema of the COQEFFECTS library



(6)

The memory module uses declarations of *locations*. A *location* represents a field on the memory to store and observe data. Then terms are defined in steps. First we give the definitions of *non-decorated terms*: they constitute the main part of the design with the inclusion of all the required functions. For instance, the `lookup` function which observes the current state is defined from void ($\mathbb{1}$, the terminal object of the underlying category) to the set of values that could be stored in that specified location. The next step is to decorate those functions with respect to their manipulation abilities on the state structure. For instance, the `update` function is defined as a *modifier*. the *modifier* status is represented by a *rw* label in the library. All the rules related to decorated functions are stated in the module called *Axioms*. Then, based on the ones already defined, some other terms are derived. For example, the derived `permut` function takes projections as the basis and replaces the orders of input objects in a *categorical product*. Similarly, by using the already defined rules (given in the *axioms* section), some additional rules are derived concerning *categorical pairs*, *products* and *others* pointing the rules constructed over the ones from different sources.

In the following subsection we detail the system sub-modules. The order of enumeration gives the dependency among sub-modules as shown in the above diagram. For instance, the module *decorations* requires

definitions from the *memory* and the *terms* modules. Then, as an example, we give the full proof, in Coq, of the update-lookup commutation property of [10].

3.1. Proof System for States

In this section we give the Coq definitions of our proof system and explore them module by module.

The major ideas in the construction of this Coq framework are:

- All the features of the proof system, that are given in the previous Sections 2.2, 2.3 and 2.4 definitely constitute the basis for the Coq implementation. In brief, we first declare all the terms without decorations, then we decorate them and after all we end up with the rules involving decorated terms. We also confirm that if one removes all the decorations (hence transforming every operations into `pure` terms), the proof system remains valid.
- The terms `pair` and `perm_pair` express the construction of pairs of two functions. As shown in Section 2.3, in the presence of effects the order of evaluation matters. Therefore, we first define the left pair in Coq and simply call it `pair` (see Section 3.3). Then, the right pair can be *derived* using the permutation rule and it is called `perm_pair` (see Section 3.6).
- The most challenging part of the design is the proof implementations of the propositions by [10], since they are quite tricky and long. We assert implementations tricky, because to see the main schema (or flowchart) of the proofs at first sight and coding them in Coq with this reasoning is quite difficult. To do so, we first sketch the related diagrams with marked equalities (*strong* or *weak*), then we convert them into some line equations, representing the main propositions to be shown. In order to do so, we use a fractional notation together with the exploited rules for each step. Eventually, Coq implementations are done by coding each step which took part in the fractional notation. From this aspect, without the fractional correspondences, proofs might be seen a little tough to follow. In order to increase the readability score, we divided those implementations into sub-steps and gave the associated relevant explanations. See Section 4 for an example.
- Considering the entire design, we benefit from an important aspect provided by Coq environment, namely *dependent types*. They provide a unified formalism in the creation of new data types and allow us to deal in a simple manner with most of the typing issues. More precisely, the type **`term`** is not a `Type`, but rather a `Type` → `Type` → `Type`. The domain/codomain information of **`term`** is embedded into Coq type system, so that we do not need to talk about ill-typed terms. For instance, `pi1 ∘ final` is ill-typed since `final` is defined from any object X: `Type` to **`unit`** where `pi1` if from Y: `Type` to Z: `Type`. Therefore, the latter composition cannot be seen as a **`term`**.

3.2. Memory

We represent the set of memory locations by a Coq parameter $Loc : Type$. Since memory locations may contain different types of values, we also assume a function $Val : Loc \to Type$ that indicates the type of values contained in each location.

3.3. Terms

Non-decorated operators, using the monadic equational logic and categorical products, are represented by an inductive Coq data type named **`term`**. It basically gets two Coq types, that are corresponding either to objects or to mappings in the given categorical structure, and returns a function type. Those function types are the representations of the homomorphisms of the category. We summarize these non-decorated constructions below:

```
Inductive term: Type → Type → Type :=
  | id: ∀ {X: Type}, term X X
  | comp: ∀ {X Y Z: Type}, term X Y → term Y Z → term X Z
  | final: ∀ {X: Type}, term unit X
  | pair: ∀ {X Y Z: Type}, term X Z → term Y Z → term (X×Y) Z
  | pi1: ∀ {X Y: Type}, term X (X×Y)
  | pi2: ∀ {X Y: Type}, term Y (X×Y)
  | lookup: ∀ i: Loc, term (Val i) unit
  | update: ∀ i: Loc, term unit (Val i).
Infix "o" := comp (at level 70).
```

Note that a term of type **term** X Y is interpreted as a function from the set Y to the set X (the co-domain, X, is given first.) The constructor `id` denotes the identity function: for any type X, `id` X has type **term** X X. The term `comp` composes two given compatible function types and returns another one. The term `pair` represents the categorical product type of two given objects. For instance, if **term** X Z corresponds to a mapping defined from an object Z to another one denoted as X, then `pair` with input types **term** X Z and **term** Y Z, agreeing on domains, returns a new function type of form **term** (X × Y) Z. The terms `pi1` and `pi2` are projections of products while `final` maps any object to the terminal object (the singleton set, denoted by $\mathbb{1}$) of the Cartesian effect category in question. `lookup` takes a location identifier and denotes the lookup operation for the relevant location. It is mathematically defined from the terminal object of the category. As the name suggests, the `update` operator updates the value in the specified location.

3.4. Decorations

In order to keep the semantics of state close to syntax, all the operations are decorated with respect to their manipulation abilities on the state structure. In Coq, we define another inductive data type, called **kind**, to represent these decorations. Its constructors are `pure` (decorated by *pure*), `ro` (for read-only and decoration *ro*) and `rw` (for read-write and decoration *rw*). It should be recalled that if a function is `pure`, then it could be seen both as `ro` (accessor) and `rw` (modifier), due to the hierarchy rule among decorated functions:

```
Inductive kind := pure | ro | rw.
```

In Coq, we had to define the decorations of terms via the separate inductive data type called **is**. The latter takes a term and a kind and returns a `Prop`. In other words, **is** indicates whether the given term is allowed to be decorated by the given kind or not. For instance, the term `id` is `pure`, since it cannot use nor modify the state. Therefore it is by definition decorated with the keyword *pure*. This decoration is checked by a constructor `is_id`. To illustrate this, if one (by using `apply` tactic of Coq) asks whether `id` is pure, then the returned result would have to be `True`. In order to check whether `id` is an accessor or a modifier, the constructors `is_pure_ro` and `is_ro_rw` should be applied beforehand to convert both statements into **is** `pure id`. The incidence of decorations upon the terms is summarized below together with their related rules:

```
Inductive is: kind → ∀ X Y, term X Y → Prop :=
 | is_id: ∀ X, is pure (@id X)
 | is_comp: ∀ k X Y Z (f: term X Y) (g: term Y Z), is k f → is k g → is k (f o g)
 | is_final: ∀ X, is pure (@final X)
 | is_pair: ∀ k X Y Z (f: term X Z) (g: term Y Z), is k f → is k g → is k (pair f g)
 | is_pi1: ∀ X Y, is pure (@pi1 X Y)
 | is_pi2: ∀ X Y, is pure (@pi2 X Y)
 | is_lookup: ∀ i, is ro (lookup i)
 | is_update: ∀ i, is rw (update i)
 | is_pure_ro: ∀ X Y (f: term X Y), is pure f → is ro f
 | is_ro_rw: ∀ X Y (f: term X Y), is ro f → is rw f
```

The decorated functions stated above are classified into four different manners: terms specific to state effect: `is_lookup` and `is_update`, categorical terms: `is_id`, `is_comp` and `is_final`, terms related to categorical

products: `is_pair`, `is_pi1` and `is_pi2` and the term decoration conversions based on the operation hierarchy: `is_pure_ro` and `is_ro_rw`.

The **term** `comp` enables one to compose two compatible functions while the constructor `is_comp` enables one to compose functions and to preserve their common decoration. For instance, if a `ro` function is composed with another `ro`, then the composite function becomes `ro` as well. For the case of the `pair`, the same idea is used. Indeed, the constructor `is_pair` takes two terms agreeing on domains such as **term** Y_1 X, say an `ro`, and **term** Y_2 X, which is `ro` as well. `is_pair` then asserts that the pair of these terms is another `ro`. It is also possible to create both compositions and pairs of functions with different decorations via the hierarchy rule stated among decoration types. This hierarchy is build via the last two constructors, `is_pure_ro` and `is_rp_rw`. The constructor `is_pure_ro` indicates the fact that if a term is `pure`, then it can be seen as `ro`. Lastly `is_ro_rw` states that if a term is `ro`, then it can be seen as `rw` as well. Note that the details of building pairs with different decorations can be found in the derived pairs module (`Pairs.v` in the library). The terms `final`, `pi1` and `pi2` are all `pure` functions since they do not manipulate the state. `final` forgets its input argument(s) and returns nothing. Although this property could make one think that it generates a sort of side-effect, this is actually not the case. Indeed, it is the only pure function whose co-domain is the terminal object ($\mathbb{1}$) and it is therefore used to simulate the execution of a program: successive, possibly incompatible, functions can then be composed with this intermediate forgetfulness of results. The `lookup` functions are decorated with the keyword *ro*, as accessors. The constructor `is_lookup` is used to check the validity of the `lookup`' decoration. The different `update` functions are `rw` and decorated with the keyword *rw*. Similarly, the constructor `is_update` is thus used to check the validity of the `update`' decoration.

3.5. Axioms

This section explains the Coq implementations of the axioms or rules (used in equational proofs) based on *monadic equational logic*, *categorical* and *observational products*. As stated in the Section 2.2, the equations are decorated as well. We have the *weak* equality between parallel morphisms modeling the fact that those morphisms return the same value but may perform different manipulations of the state. If both have the same returned results with identical state manipulations, then the equality becomes *strong*. In order to define these decorations of equations in Coq, we again use inductive terms and preserve the naming strategy of Section 2. Below are given the reserved notations for *strong* and *weak* equalities, respectively.

```
Reserved Notation " x == y" (at level 80).        Reserved Notation "x ∼ y" (at level 80).
```

The rules used to construct the proof given in Section 4 are detailed below. It is also worth to note that strong and weak are *mutually* defined inductive types to enable the implementation of the cases where both of them may appear inside the same constructor. E.g., when converting between those equalities: `ro_weak_to_strong` and `strong_weak`.

```
Inductive strong: ∀ X Y, relation (term X Y) :=
```

- The rules `strong_refl`, `strong_sym` and `strong_trans` state that *strong* equality is reflexive, symmetric and transitive, respectively. Obviously, it is an equivalence relation.
- Both `id_src` and `id_tgt` state that the composition of any arbitrarily selected function with `id` is itself regardless of the composition order.
- `strong_subs` states that strong equality obeys the substitution rule. That means that for a pair of parallel functions that are *strongly* equal, the compositions of the same source compatible function with those functions are still *strongly* equal. `strong_repl` states that for those parallel and *strongly* equal function pairs, their compositions with the same target compatible function are still *strongly* equal.
- `ro_weak_to_strong` is the rule saying that all *weakly* equal `ro` terms are also *strongly* equal. Intuitively, from the given *weak* equality, they have the same results. Now, since they are not modifiers, they cannot modify the state meaning that effect equality requirement is also met.

- `comp_final_unique` ensures that two parallel `rw` functions (say f and g) are *strongly* equal if they return the same result: `f ∼ g` together with the same effect: `final ∘ f == final ∘ g`.

`with` **weak**: ∀ X Y, relation (**term** X Y) :=

- `pure_weak_repl` states that *weak* equality obeys the substitution rule stating that for a pair of parallel functions that are *weakly* equal, the compositions of those functions with the same target compatible and `pure` function are still *weakly* equal.
- `strong_to_weak` states that *strong* equality could be converted into *weak* one, free of charge. Indeed, the definition of *strong* equality encapsulates the one for *weak* equality.
- `axiom_2` states that first updating a location `i` and then implementing an observation to another location `k` is *weakly* equal to the operation which first forgets the value stored in the location `i` and observes location `k`.

Please note that *weak* equality is an equivalence relation and obeys the substitution rule such as the *strong* one.

3.6. Derived Terms

Additional to those explained in Section 3.3, some extra terms are derived via the definitions of already existing ones:

```
Definition inv_pi1 {X Y}: term (X×unit) (X) := pair id unit.
Definition permut {X Y}: term (X×Y) (Y×X) := pair pi2 pi1.
Definition perm_pair {X Y Z} (f: term Y X) (g: term Z X): term (Y×Z) X
:= permut ∘ pair g f.
Definition prod {X Y X' Y'} (f: term X X') (g: term Y Y'): term (X×Y) (X'×Y')
:= pair (f ∘ pi1) (g ∘ pi2).
Definition perm_prod {X Y X' Y'} (f: term X X') (g: term Y Y'): term (X×Y) (X'×Y')
:= perm_pair (f ∘ pi1) (g ∘ pi2).
```

`Val_i` and `Val_i×𝟙` are isomorphic. Indeed, on the one hand, let us form the left semi-pure pair $h = \langle id_{Val_i}, \langle\rangle_{Val_i}\rangle_r$. As $\langle\rangle_{Val_i}$ is pure, then so is h. Now, from the definitions of semi-pure products (see Figure 3) the projections yields $\pi_1 \circ h \sim id_{Val_i}$, which is also $\pi_1 \circ h == id_{Val_i}$ since all the terms are pure. On the other hand, $\pi_1 \circ (h \circ \pi_1) == id_{Val_i} \circ \pi_1 == \pi_1 == \pi_1 \circ (id_{Val_i \times \mathbb{1}})$ and $\pi_2 \circ (h \circ \pi_1) == \langle\rangle_{Val_i} \circ \pi_1 \sim \langle\rangle_{Val_i \times \mathbb{1}} \sim \pi_2 == \pi_2 \circ (id_{Val_i \times \mathbb{1}})$. but the latter weak equivalences are strong since all the terms are pure. Therefore $\pi_1 \circ (h \circ \pi_1) == \pi_1 \circ (id_{Val_i \times \mathbb{1}})$ and $\pi_2 \circ (h \circ \pi_1) == \pi_2 \circ (id_{Val_i \times \mathbb{1}})$ (see Equation 2) so that $h \circ \pi_1 == id_{Val_i \times \mathbb{1}}$. Overall we have that π_1 is invertible and $\pi_1^{-1} = h = \langle id_{Val_i}, \langle\rangle_{Val_i}\rangle_r$ as defined above.

We also have the `permut` term. It takes a product, switches the order of arguments involved in the input product cone and returns the new product: its signature is **term** (Y×X) (X×Y). The **term** `perm_pair f g` is handled via the composition of `pair g f` with `permut`. The definition `prod` is based on the definition of `pair` with a difference that both input functions are taking a product object and returning another one while `perm_prod` is the permuted version of `prod` which is built on `perm_pair`s.

The decorations of `perm_pair`, `prod` and `perm_prod`, depend on the decorations of their input arguments. For instance, a `perm_pair` of two `pure` functions is also `pure` while the `prod` and `perm_prod` of two `rw`s is a `rw`. These properties are provided by `is_perm_pair`, `is_prod` and `is_perm_prod`. More details can be found in the associated module of the library (`Derived_Terms.v`). Note that it is also possible to create `perm_pair`s, `prod`s and `perm_prod`s of functions with different decorations via the hierarchy rule stated among decoration types (`is_pure_ro` and `is_rp_rw`). Existence proofs together with projection rules, can also be found in their respective modules in the library (`Decorated_Pairs.v` and `Decorated_Products.v`).

3.7. Decorated Pairs

In this section we present some of the derived rules, related to *pairs* and *projections*. In Section 2.3 we have defined the left semi-pure pair $\langle id_X, f\rangle_l^{rw} : X \to X \times Y$ of the identity id_X^{pure} with a modifier $f^{rw} : X \to Y$. In Coq

this construction will be called simply the `pair` of id_X^{pure} and f^{rw}. The right semi-pure pair $\langle f, id_X \rangle_r^{rw} : X \to Y \times X$ of f^{rw} and id_X^{pure} can be obtained as $\langle id_X, f \rangle_l^{rw}$ followed by the permutation $perm_{X,Y} : X \times Y \to Y \times X$, it will be called the `perm_pair` of f^{rw} and id_X^{pure}. Then, the `pair` and `perm_pair` definitions, together with the hierarchy rules among function classes (`is_pure_ro` and `is_ro_rw`), are used to derive some other rules related to existences and projections.

- `dec_pair_exists_purerw` is the rule that ensures that a `pair` with a `rw` and a `pure` arguments also exists and is `rw` too. `weak_proj_pi1_purerw` is the first projection rule stating that the first result of the pair is equal to the result of its first coefficient function. In our terms it is given as follows: `pi1 ∘ pair f1 f2 ∼ f1`. The given equality is *weak* since its left hand side is `rw`, while its right hand side is `pure`. `strong_proj_pi2_purerw` is the second projection rule of the semi-pure pair. It states that the second result of the pair and its effect are equal to the result and effect of its second coefficient function. In our terms it is given as follows: `pi2 ∘ pair f1 f2 == f2`. `dec_perm_pair_exists_rwpure` is similar with pure and modifier inverted.

- `dec_pair_exists_purero` is similar but with one coefficient function `pure` and the other `ro`. Thus it must be an accessor by itself and its projections must be *strongly* equal to its coefficient functions, since there are no modifiers involved. These properties are stated via `strong_proj_pi1_purero` (`pi1 ∘ pair f1 f2 == f1`) and `strong_proj_pi2_purero` (`pi2 ∘ pair f1 f2 == f2`). `dec_perm_pair_exists_ropure` is similar with pure and accessor inverted.

More details can be found in the `Decorated_Pairs.v` source file.

3.8. Decorated Products

Semi-pure products are actually specific types of semi-pure pairs, as explained in Section 2.3. In the same way in Coq, the `pair` and `perm_pair` definitions give rise to the `prod` and `perm_prod` ones.

- `dec_prod_exists_purerw` ensures that a `prod` with a `pure` and a `rw` arguments exists and is `rw`. `weak_proj_pi1_purerw_rect` is the first projection rule and states that `pi1 ∘ (prod f g) ∼ f ∘ pi1`. `strong_proj_pi2_purerw_rect` is the second projection rule and assures that `pi2 ∘ (prod f g) == f ∘ pi2`. Similarly, the rule `dec_perm_prod_exists_rwpure` with projections: `strong_perm_proj_pi1_rwpure_rect` (`pi1 ∘ (perm_prod f g) == f ∘ pi1`) and `weak_perm_proj_pi2_rwpure_rect` (`pi2 ∘ (perm_prod f g) ∼ g ∘ pi2`) relate to permuted products.

- `dec_prod_exists_purero` ensures that a `prod` with a `pure` and a `ro` arguments exists and is `ro`. `weak_proj_pi1_purero_rect` is the first projection rule and states that `pi1 ∘ (prod f g) == f ∘ pi1`. `strong_proj_pi2_purerw_rect` is the second projection rule and assures that `pi2 ∘ (prod f g) == f ∘ pi2`. Similarly, permutation rule could be applied to get `dec_prod_exists_ropure` rule with its projections: `strong_perm_proj_pi1_ropure_rect` (`pi1 ∘ (perm_prod f g) == f ∘ pi1`) and `weak_perm_proj_pi2_ropure_rect` (`pi2 ∘ (perm_prod f g) == g ∘ pi2`)

For further explanation of each derivation with Coq implementation, refer to the `Decorated_Products.v` source file.

3.9. Derived Rules

The library also provides derived rules which can be just simple shortcuts for frequently used combinations of rules or more involved results. For instance:

- `weak_refl` says *weak equality* is reflexive: It is derived from the reflexivity of the *strong equality*.

- Two pure functions having the same codomain $\mathbb{1}$ must be strongly equal (no result and state unchanged). Therefore `E_0_3` extends this to a composed function $f \circ g$, for two pure compatible functions f and g, and another function h, provided that g and h have $\mathbb{1}$ as codomain.
- In the same manner, `E_1_4` states that the composition of any `ro` function h: $\mathbb{1} \to X$, with `final` is strongly equal to the `id` function on $\mathbb{1}$. Indeed, both have no result and do not modify the state.

More similar derived rules can be found in the `Derived_Rules.v` source file.

4. Implementation of a proof: update-lookup commutation

We now have all the ingredients required to prove the update-lookup commutation property of Section 2.4 the order of operations between updating a location `i` and retrieving the value at another location `j` does not matter. The formal statement is given in Equation (5). The value intended to be stored into the location `i` is an element of `Val_i` set while the lookup operation to the location `j` takes nothing (apart from `j`), and returns a value read from the set `Val_j`. If the order of operations is reversed, then the element of `Val_i` has to be preserved while the other location is examined. Thus we need to form a product with identity and lookup following the creation of the object `Val_i` $\times$ $\mathbb{1}$, via `inv_pi1`. Similarly, the value recovered by the lookup operation has to be preserved and returned after the update operation. Then a product with the identity and update is created and a projection is used to separate their results. The full Coq proof thus uses the following steps which are detailed immediately after the general sketch:

```
1.  assume i, j:Loc
2.  | lookup j ∘ update i == pi2 ∘ (perm_prod (update i) id)
3.  |  ∘ (prod id (lookup j)) ∘ inv_pi1                          by comp_final_unique
4.  | step 1
5.  | | final ∘ lookup j ∘ update i == final
6.  | | ∘ pi2 ∘ (perm_prod (update i) id)                        by strong_sym
7.  | | ∘ (prod id (lookup j)) ∘ inv_pi1
8.  | | final ∘ pi2 ∘ (perm_prod (update i) id)
9.  | | ∘ (prod id (lookup j)) ∘ inv_pi1
10. | |  == final ∘ lookup j ∘ update i                          by strong_trans
11. | | substep 1.1
12. | | | final ∘ pi2 ∘ (perm_prod (update i) id)
13. | | | ∘ (prod id (lookup j)) ∘ inv_pi1
14. | | | == pi1 ∘ (perm_prod (update i) id)
15. | | | ∘ (prod id (lookup j)) ∘ inv_pi1                      by E_0_3
16. | | substep 1.2
17. | | | pi1 ∘ (perm_prod (update i) id)
18. | | | ∘ (prod id (lookup j)) ∘ inv_pi1
19. | | | == update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1    by strong_perm_proj
20. | | substep 1.3
```

```
21.       update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1
22.       == update i ∘ pi1 ∘ inv_pi1                          by strong_proj
23.     substep 1.4
24.       update i ∘ pi1 ∘ inv_pi1 == update i ∘ id            by id_tgt
25.     substep 1.5
26.       final ∘ lookup j ∘ update i == update i ∘ id         by E_1_4
27.   step 2
28.     lookup j ∘ update i ∼ pi2 ∘ (perm_prod (update i) id)
29.     ∘ (prod id (lookup j)) ∘ inv_pi1                       by weak_trans
30.     substep 2.1
31.       lookup j ∘ update i ∼ lookup j ∘ final               by axiom_2
32.     substep 2.2
33.       lookup j ∘ final ∼ lookup j ∘ pi2 ∘ inv_pi1          see § 3.7
34.     substep 2.3
35.       pi2 ∘ (perm_prod (update i) id)
36.       ∘ (prod id (lookup j)) ∘ inv_pi1
37.       ∼ pi2 ∘ (prod id (lookup j)) ∘ inv_pi1               by strong_perm_proj
38.     substep 2.4
39.       pi2 ∘ (prod id (lookup j)) ∘ inv_pi1
40.       ∼ lookup j ∘ pi2 ∘ inv_pi1                           by strong_proj
41. lookup j ∘ update i == pi2 ∘ (perm_prod (update i) id)
42. ∘ (prod id (lookup j)) ∘ inv_pi1
```

To prove such a proposition, the comp_final_unique rule is applied first and results in two sub-goals to be proven: `final ∘ lookup j ∘ update i == final ∘ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1` (to check if both hand sides have the same effect or not) and `lookup j ∘ update i ∼ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1` (to see whether they return the same result or not). Proofs of those sub-goals are given in step 1 and step 2, respectively.

Step 1. `final ∘ lookup j ∘ update i == final ∘ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1`:

(1.1) The left hand side `final ∘ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1`, (after the strong_sym rule application) is reduced into: `pi1 ∘ (perm_prod (update i) id) ∘ (prod (id (Val i)) (lookup j)) ∘ inv_pi1`. The base point is an application of E_0_3, stating that `final ∘ pi2 == pi1` and followed by strong_subs applied to `(perm_prod (update i) id)`, `(prod id (lookup j))` and `inv_pi1`.

(1.2) In the second sub-step, strong_perm_proj_pi1_rwpure_rect rule is applied to indicate the strong equality between `pi1 ∘ perm_prod (update i) id` and `update i ∘ pi1`. After the applications of strong_subs with arguments `prod id (lookup j)` and `inv_pi1`, we get: `pi1 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1 == update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1`. Therefore, the left hand side of the equation can now be stated as: `update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1`.

(1.3) Then, the third sub-step starts with the application of weak_proj_pi1_purerw_rect rule in order to express the following weak equality: `pi1 ∘ prod id (lookup j) ∼ id ∘ pi1`. The next step is converting the existing weak equality into a strong one by the application of

ro_weak_to_strong, since none of the components are modifiers. Therefore we get: pi1 ∘ prod id (lookup j) == id ∘ pi1. Now, using id_tgt, we remove id from the right hand side. The subsequent applications of strong_subs with arguments inv_pi1 and strong_repl enables us to relate update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1 with update i ∘ pi1 ∘ inv_pi1 via a strong equality.

(1.4) In this sub-step, update i ∘ pi1 ∘ inv_pi1 is simplified into update i ∘ id. To do so, we start with strong_proj_pi1_purepure so that pi1 ∘ pair id final == id, where pair id final defines inv_pi1=pair id final. Then, the application of strong_repl to update i provides: update i ∘ pi1 ∘ pair id final == update i ∘ id.

(1.5) In the last sub-step, the right hand side of the equation, final ∘ lookup j ∘ update i, is reduced into update i ∘ id. To do so, we use E_1_4 which states that final ∘ lookup j == id. Then, using strong_subs on update i, we get: final ∘ lookup j ∘ update i == id ∘ update i. By using id_tgt again we remove id on the right hand side and id_src rewrites final ∘ lookup j ∘ update i as update i ∘ id.

At the end of the third step, the left hand side of the equation is reduced into the following form: update i ∘ id via a strong equality. Thus, in the fourth step, it was sufficient to show final ∘ lookup j ∘ update i == update i ∘ id to prove final ∘ pi2 ∘ (perm_prod (update i) id)) ∘ (prod id (lookup j)) ∘ inv_pi1 == final ∘ lookup j ∘ update i. This shows that both sides have the same effect on the state structure.

Step 2. We now turn to the second step of the proof, namely: lookup j ∘ update i ∼ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1. The results returned by both input composed functions are examined. Indeed, from step 1 we know that they have the same effect and thus if they also return the same results, then they we will be strongly equivalent.

(2.1) Therefore, the first sub-step starts with the conversion of the left hand side of the equation, lookup j ∘ update i, into lookup j ∘ final via a weak equality. This is done by the application of the axiom_2 stating that lookup j ∘ update i ∼ lookup j ∘ final for j ≠ i.

(2.2) The second sub-step starts with the application of strong_proj_pi2_purepure which states pi2 ∘ (pair id final) == final still with (pair id final) = inv_pi1. Then, via the applications of strong_repl, with argument lookup j, strong_to_weak and strong_sym, we get: lookup j ∘ final ∼ lookup j ∘ pi2 ∘ inv_pi1.

(2.3) In the third sub-step, the right hand side of the equation, pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1, is simplified by weak equality: we start with the application of weak_perm_proj_pi2_rwpure_rect since pi2 ∘ (perm_prod (update i) id) ∼ id ∘ pi2. Then, we once again use id_tgt to remove the identity and the applications of weak_subs with arguments prod id (lookup j) and inv_pi1 yields the following equation: pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1 ∼ pi2 ∘ (prod id (lookup j)) ∘ inv_pi1.

(2.4) In the last sub-step, pi2 ∘(prod id (lookup j)) ∘ inv_pi1 is reduced into lookup j ∘ pi2 ∘ inv_pi1 via a weak equality using strong_proj_pi2_purerw_rect so that pi2 ∘ (prod id (lookup j)) == lookup j ∘ pi2. Then, strong_repl is applied with argument inv_pi1. Finally, the strong_to_weak rule is used to convert the strong equality into a weak one.

Both hand side operations return the same results so that the statement lookup j ∘ update i ∼ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1 is proven.

Merging the two steps (same effect and same result) yields the proposition that both sides are strongly equal. The full Coq development can be found in the library in the source file Proofs.v. □

5. Conclusion

In this paper, we introduced a framework in the Coq proof assistant in order to enable programmers to verify properties of programs involving the state effect. We then used this framework to verify several well known properties of the state. This framework has been also successfully used to check a more involved proof, namely that of Hilbert-Post completeness of the global state effect in a decorated setting [6]. The process of writing this proof in our Coq environment (now more than 16 Coq pages) helped discovering one non obvious flaw in a preliminary version of the proof. Future work includes extending this framework for the state in order to deal with memory allocation, conditionals and loops. We also plan to write a framework for the *exception* effect: we know from [4] that the *core* part of exceptions is dual to the global state effect. Then these frameworks will be merged for dealing with the *composition* of effects. In order to write frameworks in Coq for more general monadic or comonadic effects, the generic patterns of [6] should be helpful.

References

[1] B. Ahrens and J. Zsido. Initial semantics for higher-order typed syntax. *Journal of Formalized Reasoning*, 4(1), 2011. http://arxiv.org/abs/1012.1010.

[2] Y. Bertot. *From Semantics to Computer Science, essays in Honour of Gilles Kahn*, chapter Theorem proving support in programming language semantics, pages 337–361. Cambridge University Press, 2009. http://hal.inria.fr/inria-00160309/.

[3] J.-G. Dumas, D. Duval, L. Fousse, and J.-C. Reynaud. Decorated proofs for computational effects: States. In U. Golas and T. Soboll, editors, *ACCAT*, volume 93 of *EPTCS*, pages 45–59, 2012. http://hal.archives-ouvertes.fr/hal-00650269.

[4] J.-G. Dumas, D. Duval, L. Fousse, and J.-C. Reynaud. A duality between exceptions and states. *Mathematical Structures in Computer Science*, 22(4):719–722, Aug. 2012.

[5] J.-G. Dumas, D. Duval, and J.-C. Reynaud. Cartesian effect categories are freyd-categories. *J. Symb. Comput.*, 46(3):272–293, 2011. http://hal.archives-ouvertes.fr/hal-00369328.

[6] J.-G. Dumas, D. Duval, and J.-C. Reynaud. Patterns for computational effects arising from a monad or a comonad. Technical report, IMAG-hal-00868831, arXiV cs.LO/1310.0605, Oct. 2013.

[7] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[8] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In J. Ferrante and P. Mager, editors, *POPL*, pages 47–57. ACM Press, 1988.

[9] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

[10] G. D. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[11] J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical. Structures in Comp. Sci.*, 7(5):453–468, Oct. 1997.

[12] G. Stewart. Computational verification of network programs in coq. In *Proceedings of Certified Programs and Proofs (CPP 2013), Melbourne, Australia*, Dec. 2013. http://www.cs.princeton.edu/~jsseven/papers/netcorewp.

[13] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003.

Vérification de programmes C concurrents avec Cubicle : Enfoncer les barrières

Sylvain Conchon[1,2] & David Declerck[1] & Luc Maranget[3] & Alain Mebsout[1,2]

1: Université Paris Sud, CNRS, F-91405 Orsay
2: INRIA Saclay – Île-de-France, F-91893 Orsay
3: INRIA Paris – Rocquencourt, F-78153 Le Chesnay
sylvain.conchon@lri.fr, david.declerck@u-psud.fr
luc.maranget@inria.fr, alain.mebsout@lri.fr

Résumé

Toutes les bibliothèques de *threads* au standard POSIX se doivent d'implémenter une barrière de synchronisation. Une telle structure de contrôle permet à des *threads* de s'attendre en un point donné d'un programme. Il existe de nombreuses implémentations pour ces barrières, plus ou moins sophistiquées. Citons par exemple, les *sense barriers*, les *static tree barriers*, les *tournament barriers*, etc. Nous présentons dans cet article un outil pour vérifier automatiquement la sûreté de barrières de synchronisation écrites en langage C. Pour être sûre, une barrière doit garantir qu'aucun *thread* ne peut la franchir tant que les autres ne l'ont pas atteinte, et ceci quelque soit le nombre de *threads* impliqués. Notre approche consiste à compiler des programmes C avec *threads* vers des systèmes de transition paramétrés, puis à vérifier leur sûreté à l'aide d'un *model checker*. Plus concrètement, notre compilateur traduit un sous-ensemble du C vers le langage d'entrée de Cubicle. On montre le bien fondé de notre technique par la preuve de sûreté de la barrière *sense reversing* pour un nombre quelconque de *threads*. À notre connaissance, c'est la première preuve automatique d'une telle implémentation.

1. Introduction

La programmation d'applications concurrentes avec processus légers (*threads*) repose habituellement sur l'utilisation de bibliothèques spécialisées. Par exemple, si on programme en langage C sous Linux, BSD ou Mac OS, on peut utiliser les bibliothèques NPTL ou GNU Pth qui implémentent la norme POSIX IEEE 1003.1c-1995 [16]. Cette norme définit une interface pour créer et ordonnancer des *threads*, gérer des *mutexes*, des signaux, et fournit également des mécanismes de synchronisation. Parmi ces mécanismes, on trouve les barrières.

Une barrière de synchronisation est une technique qui permet à des *threads* de se retrouver en un point donné du programme. Chaque *thread* qui participe à la barrière attend tous les autres. Lorsque le dernier arrive, tous les *threads* participant reprennent leur exécution. La norme POSIX offre une implémentation des barrières mais nous allons définir et prouver la nôtre. On définit donc un type des barrières (par exemple `barrier_t`). À cette structure de données on associe une fonction `barrier_init` pour l'initialiser avec un certain nombre de participants, ainsi qu'une fonction de synchronisation `barrier_wait`.

La conception d'une barrière de synchronisation est délicate. En effet, en plus des considérations habituelles liées à l'utilisation de verrous (attente active ou passive, réutilisation, opérations bloquantes, *etc.*), une bonne implémentation de barrière doit également minimiser le temps entre l'arrivée du dernier *thread* et la sortie de la barrière du dernier *thread*. Elle doit également faire

en sorte que tous les *threads* quittent la barrière à peu près au même moment. Ainsi, on trouve de nombreux algorithmes dans la littérature comme par exemple, les barrières centralisées (*sense* et *sense-reversing barrier*), les barrières arborescentes (*static tree* et *combining tree barrier*) ou les barrières à tableaux (*dissemination* et *tournament barrier*) [15]. Bien évidemment, en plus de ces considérations, il est primordial que l'implémentation d'une barrière soit sûre, c'est-à-dire qu'elle garantisse qu'aucun *thread* ne quitte la barrière avant l'arrivée des autres. De ce fait, la programmation de barrières de synchronisation est une tâche difficile, sachant que le débuggage de code concurrent est un cauchemar.

Le problème qui nous intéresse dans cet article est de vérifier formellement et automatiquement la sûreté, c'est-à-dire la synchronisation correcte, de barrières écrites en langage C. Les barrières de synchronisation étant conçues indépendamment du nombre de *threads* pour lesquelles elles seront utilisées, il advient donc de les vérifier pour un nombre quelconque de participants, *i.e.* de façon *paramétrée*.

D'une manière générale, la vérification de programmes concurrents est une tâche ardue. Cette activité remonte aux années 70, avec des travaux de recherche sur des extensions des méthodes manuelles de Hoare et de Floyd [5, 25], et au début des années 80 avec des approches automatiques par *model checking* [6, 27]. En ce qui concerne les programmes C concurrents, leur vérification est un domaine de recherche toujours très actif [7, 11, 17, 23, 24].

Dans cet article, on propose une nouvelle approche qui consiste à compiler des programmes C avec *threads* vers des systèmes de transition paramétrés, puis à vérifier leur sûreté à l'aide du *model checker* Cubicle [8]. Pour illustrer notre propos, on utilise l'implémentation d'une barrière *sense-reversing* présentée en section 2.

Nos contributions sont les suivantes :

- Un schéma de traduction d'un sous-ensemble du langage C concurrent vers des systèmes de transition à tableaux (section 4) dont les gardes et les actions sont décrites par des formules logiques du premier ordre. Ce fragment du langage C est limité (voir section 3) mais permet d'analyser de réelles barrières de synchronisation.
- Une analyse statique par typage pour déterminer si une variable de type `int` est utilisée comme une variable booléenne ou un compteur de *threads* (section 5). Ces informations sont nécessaires pour un traitement efficace par Cubicle.

On montre (section 6) que notre outil permet de vérifier plusieurs variantes de *sense-reversing barrier*. À notre connaissance, c'est la première preuve automatique de telles implémentations. Pour obtenir ces résulats, on a étendu l'algorithme d'atteignabilité arrière de Cubicle pour gérer plus efficacement les quantificateurs universels dans les gardes.

2. La barrière de synchronisation *Sense-Reversing*

Un exemple très simple d'utilisation d'une barrière de synchronisation en langage C est présenté dans la figure 1. La fonction principale de ce programme commence par initialiser une barrière de synchronisation avec un nombre `N` de participants, puis démarre `N` *threads* qui exécutent la fonction `runner`. Chaque *thread* entre dans une boucle infinie qui commence par un appel à `barrier_wait` pour attendre tous les autres *threads*. Après la synchronisation, le *thread* affiche son identifiant `id` puis attend à nouveau tous les autres. Quand cette deuxième synchronisation a eu lieu, seul le *thread* dont l'identifiant est 0 affiche un retour chariot.

Si la barrière de synchronisation fonctionne correctement, c'est-à-dire si aucun *thread* ne peut quitter la barrière avant l'arrivée du dernier, le comportement attendu du programme est d'afficher (à l'infini) des lignes contenant une et une seule fois chaque chiffre entre 0 et `N`-1, dans un ordre indéterminé, comme par exemple dans la figure 2. Si au contraire, la barrière est incorrecte, le programme pourra afficher des lignes quelconques, comme dans la figure 3. Une autre manière de

```c
#include <stdio.h>
#include <pthread.h>

#define N 10

barrier_t b;

void * runner(void *id) {
  while (1) {
    /// SAFETY MARK 1
    barrier_wait(&b);
    printf(" %i ", *(int *)id);
    fflush(stdout);
    /// SAFETY MARK 2
    barrier_wait(&b);
    if (*(int *) id == 0) printf("\n");
  }
  return 0;
}
```

```c
int main()
{
  int k;
  pthread_t th[N];
  int tid[N];

  barrier_init (&b, N);
  for (k = 0; k < N; k++) {
    tid[k] = k;
    pthread_create(&(th[k]), NULL, runner, &tid[k]);
  }

  for (k = 0; k < N; k++) {
    pthread_join(th[k], NULL);
  }

  return 0;
}
```

FIGURE 1 – Exemple d'utilisation d'une barrière de synchronisation

décrire la propriété de sûreté d'une barrière est qu'il ne peut y avoir deux *threads* tels que l'un soit au point de programme indiqué par le commentaire `/// SAFETY MARK 1`, et l'autre au point indiqué par le second commentaire `/// SAFETY MARK 2`.

```
9 0 6 7 3 4 8 2 1 5
3 5 8 2 7 6 0 4 9 1
0 7 8 6 1 4 2 3 5 9
8 2 1 6 5 0 3 7 4 9
6 3 2 8 5 7 4 0 9 1
3 8 4 7 5 1 9 0 2 6
4 1 2 3 7 9 0 6 8 5
4 2 1 0 3 7 5 6 8 9
2 4 5 7 0 6 9 8 3 1
9 7 1 3 0 4 5 6 2 8
...
```

FIGURE 2 – Trace correcte

```
3 1 6 4 7 5 9 8 0
2 2 9 4 1 8 7 3 6 5 0
0
7 6 2 1 8 9 4 5 3 3 0
2 8 5 3 1 4 6 9 0
2 9 1 6 7 0 2 4
8 3 5 5 9 8 0 2 4
6 1 3 7 7 9 8 3 0
5 2 6 4 1 1 0
8 4 3 5 6 2 9 7 7 9 0
...
```

FIGURE 3 – Trace incorrecte

Intéressons nous maintenant à l'implémentation de la barrière, à savoir le type `barrier_t`, et les fonctions `barrier_init` et `barrier_wait`. Parmi les nombreuses structures existantes, nous proposons d'étudier dans cet article une barrière *sense reversing* [14]. L'implémentation présentée en figure 4 est une version légèrement modifiée de la barrière proposée par Herlihy et Shavit dans « The art of Multiprocessor Programming » [15, Sec 17.3]. La modification porte sur la suppression d'une variable « *thread specific* » qui simplifie l'utilisation des barrières. Elle est suffisamment conséquente pour que la présomption de correction qui s'applique à un algorithme provenant d'un texte de référence ne se transmette pas automatiquement à sa version modifiée.

Le type `barrier_t` est un enregistrement composé de trois champs `n`, `count` et `sense`. Ces trois champs sont initialisés par la fonction `barrier_init`. Le champ `n` contient le nombre de *threads*

```
#define DECR(x) __sync_add_and_fetch(x, -1)
#define FENCE __sync_synchronize()

typedef struct barrier_t {
  unsigned int n;
  volatile unsigned int count;
  volatile int sense;
} barrier_t;

void barrier_init(barrier_t *b, unsigned int n) {
  b->n = n;
  b->count = n;
  b->sense = 0;
}
```

```
void barrier_wait(barrier_t *b) {
  int sense, rem;
  FENCE; sense = b->sense;
  FENCE; rem = DECR(&b->count);
  if (rem == 0) {
      FENCE; b->count = b->n;
      FENCE; b->sense = !sense;
  } else {
      FENCE; while (b->sense == sense);
  }
  FENCE;
}
```

FIGURE 4 – Implémentation d'une barrière *sense reversing*

impliqués dans la barrière. L'entier `count` indique le nombre de *threads* à attendre. Enfin, le booléen `sense` (de type `int` car le type `bool` n'existe pas en C) indique le *sens* de la barrière. Ce booléen est utilisé non seulement pour la synchronisation des *threads*, mais également pour une réutilisation efficace de la barrière pour des synchronisations successives. Le compteur `count` est décrémenté à chaque appel de la fonction `barrier_wait` à l'aide d'une instruction atomique (notée `DECR`). Il est réinitialisé, et le sens de la barrière est inversé, si le *thread* qui a appelé cette fonction est le dernier entré dans la barrière. Dans le cas contraire, le *thread* est mis en pause dans une boucle qui attend que le sens de la barrière s'inverse.

Le code proposé suppose que la machine sur laquelle il s'exécutera suit le modèle "*Sequential Consistency*" (SC) [19]. Dans ce modèle l'exécution parallèle résulte de l'enchevêtrement des instructions des *threads* exécutées dans l'ordre du programme et les écritures dans la mémoire prennent effet immédiatement. Or les machines modernes ne suivent pas ce modèle, mais des modèles plus relâchés, qui autorisent entre autres l'exécution des instructions dans le désordre, les tampons en écriture etc. — Voir [2] pour une introduction récente au sujet. En pratique, on peut forcer la machine cible à se comporter selon le modèle SC en insérant des instructions spécifiques, dites barrières mémoire [1], entre chaque paire d'accès à des cases distinctes de la mémoire partagée. Le code de `barrier_wait` de la figure 4 montre un placement sûr de ces barrières mémoire (notées `FENCE`). L'optimisation du placement des barrières mémoire sort du cadre de cet article — Voir par exemple [4, 21].

3. Langages source et cible

Notre approche consiste à traduire les programmes C vers des systèmes de transition paramétrés à tableaux [12]. Ces systèmes sont parfaitement adaptés à la modélisation de programmes concurrents avec un nombre quelconque de *threads*, même si leur expressivité est limitée pour permettre une vérification efficace de leurs propriétés de sûreté par *model checking*.

Systèmes de transition à tableaux

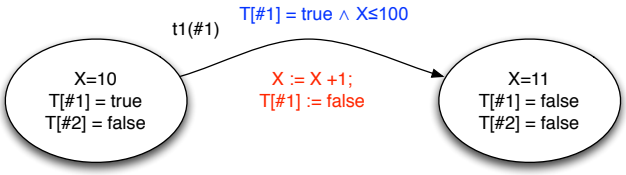
Notre langage cible est celui du *model checker* Cubicle [10]. L'état d'un système de transition est décrit par un ensemble de variables globales et de tableaux infinis indicés par des identificateurs de *thread*. Les types à notre disposition sont les entiers (`int`), les booléens (`bool`), les énumérations

1. à ne pas confondre avec les barrières de synchronisation

et le type des *threads* (`proc`). Pour décrire les transitions, on utilise dans cet article les notations de [26]. Chaque transition est représentée par une formule logique qui relie les valeurs des variables d'état avant et après la transition. Ainsi, on écrit $\mathtt{X}'$ la valeur de la variable `X` après exécution de la transition. Par exemple, si l'état d'un système est représenté par une variable entière `X` et un tableau `T`, la formule

$$t_1 : \quad \exists i.\ \mathtt{T}[i] = \mathsf{true} \wedge \mathtt{X} \leq 100 \wedge \mathtt{X}' = \mathtt{X} + 1 \wedge \mathtt{T}'[i] = \mathsf{false}$$

décrit une transition paramétrée (par i) applicable s'il existe un *thread* i tel que $\mathtt{T}[i]$ est vrai et que la valeur de `X` est inférieure à 100. Dans ce cas, l'action de cette transition est d'incrémenter la valeur de `X` et de changer la valeur de la case i de `T` à false. Graphiquement, l'application de cette transition à un système avec deux *threads* est représentée par le schéma suivant :



L'état initial d'un tel système paramétré est défini par une formule logique qui décrit les valeurs des variables globales et des tableaux pour tous les indices, *i.e.* pour tous les *threads*. Par exemple, on décrit l'état initial d'un système où `X` vaut 0 et `T` contient false pour tous les *threads* de la manière suivante :

$$\forall i.\ \neg\mathtt{T}[i] \wedge \mathtt{X} = 0$$

Les propriétés de sûreté de ces systèmes de transition sont exprimées sous forme négatives, comme des formules caractérisant les états dangereux. Par exemple, la formule suivante

$$\exists i\, j.\ i \neq j \wedge \mathtt{X} \geq 1 \wedge \mathtt{T}[i] = \mathsf{false} \wedge \mathtt{T}[j] = \mathsf{false}$$

exprime que les mauvais états du système sont ceux où il existe deux *threads* i et j distincts, tels que $\mathtt{T}[i]$ et $\mathtt{T}[j]$ sont faux et `X` est plus grand que 1.

Fragment du langage C supporté

L'expressivité de notre langage cible étant limitée, notre outil accepte un fragment volontairement restreint du langage C, mais suffisant pour écrire l'implémentation de la barrière de synchronisation donnée en figure 4 et son utilisation en figure 1.

Ainsi, les programmes acceptés ne peuvent utiliser que les types `int` et `void` (les qualificateurs `volatile` et `unsigned` sont acceptés mais ignorés). Les déclarations de structures non récursives sont également possibles. L'utilisation de pointeurs est restreinte au passage d'arguments par référence, et au retour de type `void *`. Les opérateurs et relations sur ces types sont les suivants :

- les opérations `+`, `-`, `*` et `!` ;
- les relations (`==`, `<`, `<=`, `>` et `>=`) ;
- l'accès aux champs d'une structure (`.` et `->`) ;
- les opérateurs d'adresse et de déréférencement (`&` et `*`).

Le jeu d'instructions se résume aux affectations (`=`, `++`, `--`), aux conditionnelles (avec et sans `else`), aux boucles `while` et `for`, ainsi qu'aux instructions `return` et `assert`. Les fonctions d'entrée-sortie `printf` et `fflush` sont acceptées mais ignorées dans notre schéma de compilation.

4. Schéma de compilation

Les programmes C que l'on considère sont composés d'un *thread* principal et (éventuellement) de *threads* enfants démarrés par celui-ci.

Modèle mémoire

Notre modèle mémoire pour ces programmes est très simple. L'état d'un programme est uniquement défini par un ensemble de variables globales (partagées entre tous les *threads*) et le pointeur d'instruction et la pile de chaque *thread*. Pour simplifier le schéma de traduction, les variables locales d'un *thread* sont directement stockées dans sa pile. Pour des raisons d'efficacité, on isole les valeurs de type `int` utilisées de fait comme des booléens (cf. section 5), en séparant la pile d'un *thread* en une pile d'entiers et une pile de booléens.

En ce qui concerne les déclarations globales, la traduction des variables de type `int` est immédiate : pour chaque variable `x`, on utilise une variable `GLOBAL_x`. Les structures sont quant à elles traduites *à plat*, à l'aide d'une variable par champ (ou champ de champ dans le cas de structures imbriquées). Par exemple, la variable globale `x` définie de la manière suivante

```
struct t1 { int a; int b;} ;
struct t2 { int c; struct t1 s;} ;

struct t2 x;
```

est transformée en trois variables globales `STRUCT_x_c`, `STRUCT_x_s_a` et `STRUCT_x_s_b` de type `int`. Une limitation de ce modèle *à plat* est qu'il nous conduit, pour des raisons de simplicité, à interdire l'affectation de structures (il faut passer par l'affectation manuelle de chaque champ). Il nous pousse également à restreindre le passage de structures par adresse, plutôt que par valeur, et à interdire de renvoyer des structures comme résultat d'appel de fonction.

Le pointeur d'instruction du *thread* principal est matérialisé par une variable globale `PC_M`, et celui du *thread* enfant i par l'élément d'indice i d'un tableau `PC` (*i.e.*, `PC[`i`]`). Le type de ces pointeurs est une énumération d'étiquettes de la forme

$$\texttt{type t} \;=\; \mathsf{Idle} \mid \mathsf{End} \mid \mathsf{L}_1 \mid \mathsf{L}_2 \mid \ldots$$

où les constructeurs (étiquettes) L_k représentent les points de programme des *threads*, auxquels on ajoute les deux constructeurs Idle et End, pour indiquer respectivement qu'un *thread* n'est pas démarré ou qu'il est terminé.

Notre langage cible ne permettant pas de manipuler directement des piles, on représente une pile de taille n par n variables. Par exemple, si le *thread* principal a une pile d'entiers de hauteur 2, et une pile de booléens de hauteur 3, alors les cases des piles sont représentées par les variables `STACK_M_INT_0`, `STACK_M_INT_1` et `STACK_M_BOOL_0`, `STACK_M_BOOL_1`, `STACK_M_BOOL_2`.

Les piles d'un *thread* enfant i sont représentées par des variables locales, qui sont encodées par les cases de tableaux indicés par i. Par exemple, si i a une pile d'entiers de hauteur 2 et une pile de booléens de hauteur 1, alors ses piles sont interprétées par les variables `STACK_INT_0[i]`, `STACK_INT_1[i]` et `STACK_BOOL_0[i]`.

Compilation des expressions

Dans chaque *thread*, la compilation des expressions se fait de manière récursive par la compilation des sous-expressions à l'aide des piles `STACK_INT` et `STACK_BOOL`, dont la taille maximale peut être calculée statiquement.

La compilation d'une expression produit une séquence d'*affectations atomiques* de la forme `X` $\leftarrow$ `E`, où `E` est soit une constante, une variable globale, une case de pile, ou une relation (ou opération) élémentaire entre ces dernières. Ces affectations constituent les opérations *atomiques* de notre langage source.

À titre d'exemple, pour traduire l'expression `c == (a > 1)` dans le *thread* principal, il faut générer du code qui stocke dans les piles l'évaluation des sous-expressions `c`, `1`, `a`, `a > 1` et `c == (a > 1)`. En réutilisant l'espace quand cela est possible, il suffit d'une pile d'entiers et d'une pile de booléens de taille 2 pour compiler cette expression. En prenant les variables `STACK_M_INT_0` et `STACK_M_INT_1` pour coder la pile d'entiers, et les variables `STACK_M_BOOL_0` et `STACK_M_BOOL_1` pour la pile de booléens, la compilation de l'expression précédente correspond alors aux affectations suivantes :

`c`	STACK_M_BOOL_0 $\leftarrow$ GLOBAL_c
`1`	STACK_M_INT_0 $\leftarrow$ 1
`a`	STACK_M_INT_1 $\leftarrow$ GLOBAL_a
`a>1`	STACK_M_BOOL_1 $\leftarrow$ STACK_M_INT_1 $>$ STACK_M_INT_0
`c == (a>1)`	STACK_M_BOOL_0 $\leftarrow$ STACK_M_BOOL_0 $=$ STACK_M_BOOL_1

Notons l'importance de calculer au plus juste la taille des piles pour la compilation des expressions. En effet, moins les piles sont hautes et moins le nombre de variables pour les représenter est élevé, ce qui aide considérablement la phase de *model checking* par Cubicle.

Compte tenu de la simplicité de notre modèle mémoire, les appels de fonctions sont simplement *inlinés*. Le passage de paramètres par valeur est alors simulé par l'introduction de variables intermédiaires contenant les copies des arguments. Ces variables sont allouées sur les piles `STACK_M_BOOL` et `STACK_M_INT`. Par exemple, l'appel de la fonction `fct` dans le programme

```
int a, b;

int fct(int y, int *z) { return y + *z; }

int main() { return fct(a,&b); }
```

est *inliné* de la manière suivante

```
int a, b;

int fct(int y, int *z) { return y + *z; }

int main() {
  int x;
  x = a;
  return x + b;
}
```

La traduction de l'appel de fonction correspond alors aux affectations de piles suivantes, où `STACK_M_INT_1` est utilisée comme variable intermédiaire (`x`) :

`a`	STACK_M_INT_0 $\leftarrow$ GLOBAL_a
`x = a`	STACK_M_INT_1 $\leftarrow$ STACK_M_INT_0
`b`	STACK_M_INT_2 $\leftarrow$ GLOBAL_b
`x + b`	STACK_M_INT_1 $\leftarrow$ STACK_M_INT_1 $+$ STACK_M_INT_2
`return x + b`	STACK_M_INT_0 $\leftarrow$ STACK_M_INT_1

Dans la suite, on note $\mathcal{E}(i, \texttt{exp})$ la séquence d'affectations *atomiques* qui résulte de la traduction d'une expression `exp` pour un *thread* quelconque i (principal ou enfant). Notons que la dernière affectation atomique de la séquence $\mathcal{E}(i, \texttt{exp})$ met le résultat de l'évaluation de *exp* dans le niveau 0 de la pile.

Compilation des instructions

Pour compiler les instructions du langage source, on définit une fonction $\mathcal{C}$ qui prend en arguments un identificateur de *thread*, une instruction, et deux étiquettes, et renvoie un ensemble de transitions. Ainsi, $\mathcal{C}(i, \texttt{instr}, \mathsf{L}_0, \mathsf{L}_1)$ génère le code cible pour le *thread* i qui correspond à l'exécution de `instr` depuis le point de programme L_0 jusqu'au point L_1. Ce code cible correspond à un ensemble de transitions, c'est-à-dire à un ensemble de formules comme décrites en section 3.

Affectations. La compilation d'une affectation atomique $\mathtt{X} \leftarrow \mathtt{E}$ est donnée par la transition

$$\mathcal{C}(i, \mathtt{X} \leftarrow \mathtt{E}, \mathsf{L}_0, \mathsf{L}_1) \quad = \quad \{\exists i.\ \mathtt{PC}[i] = \mathsf{L}_0 \wedge \mathtt{X}' = \mathtt{E} \wedge \mathtt{PC}'[i] = \mathsf{L}_1\}$$

qui s'applique lorsque le point de programme $\mathtt{PC}[i]$ du *thread* i est en L_0. Elle met à jour la variable `X` puis change le point de programme en L_1. Une affectation quelconque `x = exp` du langage C, où `exp` est supposée être de type `int`, se traduit alors par

$$\mathcal{C}(i, \mathtt{x} = \texttt{exp}, \mathsf{L}_0, \mathsf{L}_1) \quad = \quad \mathcal{C}(i, \mathcal{E}(i, \texttt{exp}); \mathtt{x} \leftarrow \mathtt{STACK\_INT\_0}[i], \mathsf{L}_0, \mathsf{L}_1)$$

Séquence d'instructions. La compilation d'une séquence d'instructions `instr1; instr2` est donnée par la définition

$$\begin{aligned} \mathcal{C}(i, \texttt{instr1}; \texttt{instr2}, \mathsf{L}_0, \mathsf{L}_1) \quad = \quad & \texttt{let } \mathsf{L} = \texttt{fresh\_label}() \texttt{ in} \\ & \mathcal{C}(i, \texttt{instr1}, \mathsf{L}_0, \mathsf{L}) \cup \mathcal{C}(i, \texttt{instr2}, \mathsf{L}, \mathsf{L}_1) \end{aligned}$$

qui compile récursivement l'instruction `instr1` entre le point de programme L_0 et un point de programme intermédiaire L, puis récursivement l'instruction `instr2` entre les points L et L_1.

Conditionnelles. La compilation des instructions conditionnelles nécessite quant à elle de créer trois nouvelles étiquettes L, $\mathsf{L}_{\texttt{then}}$ et $\mathsf{L}_{\texttt{else}}$. La première étiquette est utilisée pour la compilation de la condition. Les deux dernières permettent d'aiguiller les transitions vers les branches `then` ou `else`.

$$\begin{aligned} \mathcal{C}(i, \texttt{if (exp) \{ instr1 \} else \{ instr2 \}}, \mathsf{L}_0, \mathsf{L}_1) = \\ & \texttt{let } \mathsf{L} = \texttt{fresh\_label}() \texttt{ in} \\ & \texttt{let } \mathsf{L}_{\texttt{then}} = \texttt{fresh\_label}() \texttt{ in} \\ & \texttt{let } \mathsf{L}_{\texttt{else}} = \texttt{fresh\_label}() \texttt{ in} \\ & \mathcal{C}(i, \mathcal{E}(i, \texttt{exp}), \mathsf{L}_0, \mathsf{L}) \cup \\ & \{\exists i.\ \mathtt{PC}[i] = \mathsf{L} \wedge \mathtt{STACK\_BOOL\_0}[i] = \texttt{true} \wedge \mathtt{PC}'[i] = \mathsf{L}_{\texttt{then}}\ , \\ & \ \exists i.\ \mathtt{PC}[i] = \mathsf{L} \wedge \mathtt{STACK\_BOOL\_0}[i] = \texttt{false} \wedge \mathtt{PC}'[i] = \mathsf{L}_{\texttt{else}}\} \cup \\ & \mathcal{C}(i, \texttt{instr1}, \mathsf{L}_{\texttt{then}}, \mathsf{L}_1) \ \cup \ \mathcal{C}(i, \texttt{instr2}, \mathsf{L}_{\texttt{else}}, \mathsf{L}_1) \end{aligned}$$

Boucles. La compilation des boucles `while` est similaire aux conditionnelles. Elle ne nécessite que deux étiquettes intermédiaires.

$$
\begin{aligned}
&\mathcal{C}(i, \texttt{while}\,(\texttt{exp})\,\{\,\texttt{instr}\,\}, \mathsf{L}_0, \mathsf{L}_1) = \\
&\qquad \texttt{let}\ \mathsf{L} = \texttt{fresh\_label}()\ \texttt{in} \\
&\qquad \texttt{let}\ \mathsf{L}_{\texttt{while}} = \texttt{fresh\_label}()\ \texttt{in} \\
&\qquad \mathcal{C}(i, \mathcal{E}(i, \texttt{exp}), \mathsf{L}_0, \mathsf{L}) \cup \\
&\qquad \{\exists i.\ \texttt{PC}[i] = \mathsf{L} \wedge \texttt{STACK\_BOOL\_0}[i] = \texttt{true} \wedge \texttt{PC}'[i] = \mathsf{L}_{\texttt{while}}\ , \\
&\qquad\ \exists i.\ \texttt{PC}[i] = \mathsf{L} \wedge \texttt{STACK\_BOOL\_0}[i] = \texttt{false} \wedge \texttt{PC}'[i] = \mathsf{L}_1\} \cup \\
&\qquad \mathcal{C}(i, \texttt{instr1}, \mathsf{L}_{\texttt{while}}, \mathsf{L}_0)
\end{aligned}
$$

Compteurs de *threads*

Le programme d'utilisation de la barrière de synchronisation donné en figure 1 fonctionne pour un nombre de *threads* fixé à 10 par la directive :

```
# define N 10
```

Cependant, pour vérifier la propriété de bonne synchronisation de la barrière pour un nombre quelconque de *threads*, nous ne devons pas tenir compte de cette constante, mais seulement considérer `N` comme un paramètre du système.

Malheureusement, il n'est pas possible de manipuler directement ce paramètre dans notre langage cible. Seul le type `proc`, c'est-à-dire l'ensemble des indices des *threads*, permet de le représenter, `N` étant la cardinalité de cet ensemble. Puisque le domaine des tableaux est le type `proc`, on peut simplement représenter la valeur `N` comme un tableau de booléens où toutes les cases contiennent `true`. Ce tableau joue alors le rôle d'un *encodage unaire* de `N`. De la même manière, chaque variable entière `X` manipulant `N` est codée à l'aide d'un tableau de booléens. Le nombre de cases du tableau `X` ayant pour valeur `true` représente le nombre de 1 composant l'encodage unaire de `X`. On autorisera donc seulement certaines opérations sur les variables qui manipulent `N` comme un entier en C :

- l'affectation à `N` où à 0 (`Global_x = N`, `Global_x = 0`)
- le test d'égalité avec `N` et 0 (`Global_x == N`, `Global_x == 0`)
- l'incrémentation et la décrémentation atomique (*e.g.* avec la macro `DECR(&Global_x)`)

Ces variables servent donc essentiellement de compteurs de *threads*.

La compilation d'une affectation à 0 se fait en mettant toutes les cases à `false` du tableau correspondant et de manière similaire, une affectation à `N` se fait en mettant toutes les cases à `true`, de la manière suivante :

$$
\begin{aligned}
\mathcal{C}(i, \texttt{Global\_x} \leftarrow \texttt{N}, \mathsf{L}_0, \mathsf{L}_1) &= \{\exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \forall j.\ \texttt{Global\_x}'[j] = \mathsf{true} \wedge \texttt{PC}'[i] = \mathsf{L}_1\} \\
\mathcal{C}(i, \texttt{Global\_x} \leftarrow 0, \mathsf{L}_0, \mathsf{L}_1) &= \{\exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \forall j.\ \texttt{Global\_x}'[j] = \mathsf{false} \wedge \texttt{PC}'[i] = \mathsf{L}_1\}
\end{aligned}
$$

La compilation d'un test d'égalité à 0 ou à `N` sur la pile booléenne est décrite par :

$$
\begin{aligned}
&\mathcal{C}(i, \texttt{STACK\_BOOL\_}k[i] \leftarrow \texttt{Global\_x} == \texttt{N}, \mathsf{L}_0, \mathsf{L}_1) \quad = \\
&\qquad \{\exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \forall j.\ \texttt{Global\_x}[j] = \mathsf{true} \wedge \texttt{STACK\_BOOL\_}k'[i] = \mathsf{true} \wedge \texttt{PC}'[i] = \mathsf{L}_1, \\
&\qquad\ \exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \exists j.\ \texttt{Global\_x}[j] = \mathsf{false} \wedge \texttt{STACK\_BOOL\_}k'[i] = \mathsf{false} \wedge \texttt{PC}'[i] = \mathsf{L}_1\} \\
&\mathcal{C}(i, \texttt{STACK\_BOOL\_}k[i] \leftarrow \texttt{Global\_x} == 0, \mathsf{L}_0, \mathsf{L}_1) \quad = \\
&\qquad \{\exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \forall j.\ \texttt{Global\_x}[j] = \mathsf{false} \wedge \texttt{STACK\_BOOL\_}k'[i] = \mathsf{true} \wedge \texttt{PC}'[i] = \mathsf{L}_1, \\
&\qquad\ \exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \exists j.\ \texttt{Global\_x}[j] = \mathsf{true} \wedge \texttt{STACK\_BOOL\_}k'[i] = \mathsf{false} \wedge \texttt{PC}'[i] = \mathsf{L}_1\}
\end{aligned}
$$

Enfin, la traduction de la décrémentation atomique (implémentée à l'aide des instructions machine idoines, par le truchement d'un *atomic builtin* de GCC) consiste simplement à changer la valeur d'une case contenant true à false (*i.e* enlever un 1 de l'encodage unaire) :

$$\mathcal{C}(i, \texttt{DECR(\&Global\_x)}, \mathsf{L}_0, \mathsf{L}_1) = \\ \exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \exists j.\ \texttt{Global\_x}[j] = \mathsf{true} \wedge \texttt{Global\_x}'[j] = \mathsf{false} \wedge \texttt{PC}'[i] = \mathsf{L}_1$$

Propriété de sûreté

La traduction de la propriété de sûreté consiste simplement à récolter dans un ensemble S tous les points de programmes correspondant à des annotations de la forme `/// SAFETY MARK` i. On génère alors la formule suivante qui caractérise les états dangereux correspondants à ces annotations :

$$\bigvee_{\substack{(m_1, m_2) \in S \times S \\ m_1 \neq m_2}} \exists i, j.\, \texttt{PC}[i] = m_1 \wedge \texttt{PC}[j] = m_2$$

5. Typage

Il est important de pouvoir dire quand une variable entière (de type `int`) est en réalité utilisée comme un booléen pour que la phase de *model checking* effectuée par Cubicle soit la plus efficace possible. En effet le traitement de formules avec variables booléennes ne demande qu'un raisonnement propositionnel et ne fait donc appel qu'à la partie SAT du solveur SMT de Cubicle. En revanche l'utilisation d'entiers et plus particulièrement d'opérations arithmétiques demande un raisonnement spécifique de la théorie de l'arithmétique dont les appels sont coûteux.

Dans le fragment du C utilisé ici, on peut (parfois) décider statiquement si une variable est booléenne. Dans cette section on propose une analyse de typage qui, bien que limitée, donne suffisamment d'informations pour permettre la vérification de nos barrières.

Pour simplifier cette analyse, on force la distinction entre les types `bool` et `int`. Ainsi, les opérateurs arithmétiques ou de comparaison (sauf `==` et `!=`) ne pourront être utilisés que sur des valeurs de type `int`, de même, les constantes différentes de 0 et 1 seront de type `int`. Les opérateurs logiques (`&&`, `!`, etc.) seront eux exclusivement réservés aux valeurs de type `bool`. Enfin, seules les constantes 0 et 1 peuvent représenter à la fois un entier ou un booléen. Pour cette raison, on introduit des variables de type α afin de représenter le type `int`$\cup$`bool`. Regardons sur trois exemples la manière dont fonctionne notre analyse.

```
int x, y, z;

x = 0;
y = x;
z = y;
if (z) { x = x + 1; }
```

```
int x, y;

y = 0;
if (y == 0)
  { x = 0; }
```

```
int x, y, z;

x = 0 && 1;
if (x != y && y != z && x != z)
  { ... }
```

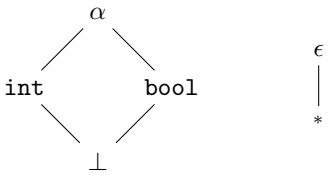
Dans le programme de gauche, juste avant l'instruction `if`, on donne le même type α aux trois variables `x`, `y` et `z`. L'analyse de la conditionnelle force alors `z` à être un booléen, tandis que `x` est utilisé comme un entier. Le programme est donc rejeté. Dans le programme du milieu, la conditionnelle force `y` a être de type `bool`, tandis que `x` reste de type α. Comme `x` peut n'avoir jamais été initialisée, et donc contenir une valeur arbitraire, on choisit de lui donner le type `int`. Enfin, dans le programme de droite, après avoir typé l'expression booléenne de la conditionnelle, on dit que `x`, `y` et `z` sont de type `bool`, mais seul `x` est initialisé. Si on donnait réellement le type `bool` à ces trois variables, la condition serait nécessairement fausse. Comme pour le programme précédent, `y` et `z` n'étant pas initialisées, elles

peuvent contenir des valeurs arbitraires et rendre la condition vraie. On donne donc le type `int` à `y` et `z` et ce programme est rejeté.

Pour simplifier notre présentation, on limite la grammaire des expressions et des instructions de notre langage source à :

$$e := x \mid \mathtt{c} \mid e{=}{=}e \mid e{+}e \mid e{<}e \mid e \,\&\&\, e$$
$$i := x{=}e \mid \mathtt{if}(e,i,i) \mid \mathtt{while}(e,i)$$

Pour faire notre analyse, on va utiliser une algèbre de type définie par $\tau := \mathtt{int} \mid \mathtt{bool} \mid \alpha$, ainsi que des marques d'*initialisation* $\mu := \epsilon \mid {}^*$. Les treillis $(\tau, \sqcup, \sqcap, \sqsubseteq)$ et $(\mu, \bigvee, \bigwedge, <)$ associés à ces ensembles sont représentés ci-dessous.



Notre algorithme de typage est défini par un ensemble de règles d'inférence dirigées par la syntaxe des expressions et des instructions. Les règles pour les expressions sont données à l'aide de séquents de la forme $\Gamma \vdash e : \tau, \mu$ signifiant « dans l'environnement Γ, l'expression e est bien typée de type τ avec une indication d'initialisation μ ». C'est seulement lorsque μ vaut * que l'on est sûr que e produit nécessairement une valeur de type τ.

Pour typer les expressions, on utilise une structure union-find sur les types dotée de deux fonctions `union` (qui réunit deux classes) et `find` (qui renvoie le représentant de la classe d'un élément). Cette structure est *globale* et utilisée par effet de bord. Elle est initialisée avec autant de classes singleton $\{\alpha_x\}$ qu'il y a de variables de terme x. De plus, elle est compatible avec le treillis des types de manière à ce que l'opération $\mathtt{union}(\tau_1, \tau_2)$ ne soit applicable que si $\tau_1 \sqcap \tau_2 \neq \bot$. Par ailleurs, cette structure doit s'assurer que le représentant d'une classe est la plus petite valeur dans le treillis. On utilise également un dictionnaire Γ qui associe chaque variable x d'un terme e à une marque μ, initialisée à ϵ. Les règles pour typer les expressions sont données ci-dessous :

$$\frac{\mathtt{c} \in \{\mathtt{0},\mathtt{1}\} \qquad \alpha \text{ est une variable fraîche}}{\Gamma \vdash \mathtt{c} : \alpha, {}^*} \qquad \frac{\mathtt{c} \notin \{\mathtt{0},\mathtt{1}\}}{\Gamma \vdash \mathtt{c} : \mathtt{int}, {}^*} \qquad \frac{}{\Gamma \vdash x : \mathtt{find}(\alpha_x), \Gamma(x)}$$

$$\frac{e_1 : \tau_1, \mu_1 \qquad e_2 : \tau_2, \mu_2 \qquad \mathtt{union}(\tau_1, \tau_2)}{\Gamma \vdash e_1{=}{=}e_2 : \mathtt{bool}, {}^*}$$

$$\frac{e_1 : \tau_1, \mu_1 \qquad e_2 : \tau_2, \mu_2 \qquad \mathtt{union}(\tau_1, \mathtt{int}) \qquad \mathtt{union}(\tau_2, \mathtt{int})}{\Gamma \vdash e_1{+}e_2 : \mathtt{int}, {}^*}$$

$$\frac{e_1 : \tau_1, \mu_1 \qquad e_2 : \tau_2, \mu_2 \qquad \mathtt{union}(\tau_1, \mathtt{int}) \qquad \mathtt{union}(\tau_2, \mathtt{int})}{\Gamma \vdash e_1{<}e_2 : \mathtt{bool}, {}^*}$$

$$\frac{e_1 : \tau_1, \mu_1 \qquad e_2 : \tau_2, \mu_2 \qquad \mathtt{union}(\tau_1, \mathtt{bool}) \qquad \mathtt{union}(\tau_2, \mathtt{bool})}{\Gamma \vdash e_1 \,\&\&\, e_2 : \mathtt{bool}, {}^*}$$

Les règles pour typer les instructions sont données plus bas à l'aide de séquents de la forme $\Gamma \vdash_i i : \Gamma'$, signifiant « dans l'environnement Γ, l'instruction i est bien typée et produit les nouvelles marques d'initialisation Γ' ». Pour gérer les marques d'initialisation, on définit la fonction `merge` sur

les dictionnaires telle que $\texttt{merge}(\Gamma_1, \Gamma_2)$ renvoie le dictionnaire qui, pour chaque couple clé-valeur $\alpha \mapsto \mu_1$ de Γ_1 et $\alpha \mapsto \mu_2$ de Γ_2, associe α à la valeur $\mu_1 \bigvee \mu_2$.

$$\frac{\Gamma \vdash e : \tau, \mu \qquad \texttt{union}(\tau, \texttt{find}(\alpha_x))}{\Gamma \vdash_i x = e : \Gamma \cup \alpha_x \mapsto \mu} \qquad \frac{\Gamma \vdash e : \tau, \mu \qquad \texttt{union}(\tau, \texttt{bool}) \qquad \Gamma \vdash_i i : \Gamma'}{\Gamma \vdash_i \texttt{while}(e, i) : \texttt{merge}(\Gamma, \Gamma')}$$

$$\frac{\Gamma \vdash e : \tau, \mu \qquad \texttt{union}(\tau, \texttt{bool}) \qquad \Gamma \vdash_i i_1 : \Gamma_1 \qquad \Gamma \vdash_i i_2 : \Gamma_2}{\Gamma \vdash_i \texttt{if}(e, i_1, i_2) : \texttt{merge}(\Gamma_1, \Gamma_2)}$$

À la fin de cette analyse, les types des variables peuvent être extraits des informations présentes dans la structure d'union-find et dans le dictionnaire Γ. Si pour une variable x, $\Gamma(x) = \epsilon$, alors la variable x n'a pas été initialisée. Son type, n'étant pas contraint, sera donc `int`. Si au contraire $\Gamma(x) = {}^*$ et $\texttt{find}(\alpha_x)$ est différent de `int`, la variable x a été initialisée et n'a pu être affectée qu'aux valeurs 0 ou 1, alors son type sera `bool`. Ces nouvelles informations de typage sont propagées dans la structure union-find afin de vérifier leur cohérence.

6. Expérimentations

On présente dans cette section les expérimentations réalisées sur différentes barrières de synchronisation. Pour toutes ces barrières on vérifie la propriété de bonne synchronisation.

On a testé notre outil sur cinq versions de barrières *sense-reversing* : `sb_alt.c` est une version avec deux fonctions `wait` alternantes, `sb.c` est la version proposée dans [15], `sb_nice.c` est la version présentée en section 2 comportant notre modification, et `sb_single.c` est une version non réutilisable (cette même barrière est utilisée – à tort – deux fois dans `sb_single_us.c`, d'où l'erreur), enfin la version `sb_loop.c` présentée plus loin est une utilisation répétée dans une boucle infinie de la barrière `sb.c`. Pour chacune, on donne les temps d'exécution de Cubicle avec et sans l'analyse de typage faite sur les booléens (présentée en section 5). On précise aussi le nombre de nœuds visités, le nombre d'invariants inférés (Inv.) ainsi que le nombre de redémarrages de la recherche (Restarts). Tous les résultats ont été obtenus sur une machine 64 bits avec un processeur Intel® Xeon® @ 3.2 GHz et 24 Go de mémoire en lançant Cubicle de manière séquentielle et avec inférence d'invariants pour deux *threads* (option `-brab 2`). Nous renvoyons le lecteur intéressé par ce mécanisme de découverte d'invariants à [9]. On note par ☠ les exemples non-sûrs dont la propriété n'est pas vérifiée et pour lesquels Cubicle expose une trace d'erreur.

	Avec typage				Sans typage			
	nœuds	Inv.	Restarts	Temps	nœuds	Inv.	Restarts	Temps
`sb_alt`	215	152	7	**7,64s**	598	180	53	11m27s
`sb.c` [15]	331	226	10	**20,7s**	414	156	34	5m21s
`sb_nice.c`	240	106	9	**11,6s**	303	139	49	28m8s
`sb_single.c`	165	115	5	**3,11s**	174	99	54	17m44s
`sb_single_us.c`☠	810	/	0	**5,06s**	811	/	0	6,13s

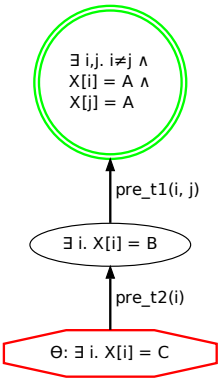
Ces résultats montrent clairement que l'analyse de typage améliore grandement l'efficacité de Cubicle. Par contre, une analyse précise des traces des expérimentations montre que le mécanisme de recherche d'invariants de Cubicle est perturbé par la présence de traces d'erreur fallacieuses. En effet, on peut se rendre compte grâce à la section 4 que la compilation des boucles et compteurs de *threads* introduit des quantificateurs universels dans les gardes. Par exemple, si `cpt` est un compteur de *threads*, la condition `cpt == 0` se traduira par $\forall i.\ \texttt{Cpt}[i] = \texttt{false}$ dans les systèmes de transition

à tableaux. Malheureusement l'utilisation de gardes dites *universelles* nous fait perdre la propriété de la décidabilité de la sûreté pour ces systèmes. En effet, les seuls moyens connus à ce jour pour traiter efficacement ces quantificateurs universels consistent tous en une forme d'*approximation*. Que ce soit dans le framework de *Regular model checking* à l'aide d'*abstractions monotones* [1] ou dans le framework *Model checking modulo theories* grâce au *Crash failure model* [3, 22], la sur-approximation introduite permet au système abstrait d'avoir des comportements qui n'existent pas dans le système réel et l'analyse pourra donc potentiellement exhiber des contre-exemples et traces d'erreur fallacieux.

Pour comprendre ce phénomène, donnons nous par exemple un tableau $\mathtt{X}$ indicé par des identificateurs de *threads* et à valeurs dans une énumération à trois constructeurs $\mathsf{A} \mid \mathsf{B} \mid \mathsf{C}$. On définit l'état initial du système par la formule $I : \forall i.\ \mathtt{X}[i] = \mathsf{A}$ et l'ensemble de ses transitions défini par :

$$\tau = \left\{ \begin{array}{lrll} t_1 : & \exists i, j. & i \neq j \wedge \mathtt{X}[i] = \mathsf{A} \wedge \mathtt{X}[j] = \mathsf{A} & \wedge\ \mathtt{X}'[i] = \mathsf{B} \\ t_2 : & \exists i. & \mathtt{X}[i] = \mathsf{B} \wedge \forall j.\ j \neq i \implies \mathtt{X}[j] \neq \mathsf{A} & \wedge\ \mathtt{X}'[i] = \mathsf{C} \end{array} \right\}$$

On veut vérifier qu'aucun état atteignable du système ne satisfait la mauvaise formule $\Theta : \exists i.\ \mathtt{X}[i] = \mathsf{C}$. Cependant, en appliquant l'algorithme classique d'atteignabilité arrière de Cubicle, on obtient successivement les nœuds représentés par l'arbre ci-dessous :



Le nœud $\exists i.\ \mathtt{X}[i] = \mathsf{B}$ est la pré-image de Θ par t_2 et indique qu'un état où il existe un *thread* i tel que $\mathtt{X}[i] = \mathsf{B}$ peut atteindre Θ en un pas. Pour calculer cet état, Cubicle a fait une approximation car il doit considérer qu'il soit possible qu'il n'y ait qu'*un seul thread* dans le programme. Ce nœud étant lui-même atteignable en un pas par l'état initial I, on obtient la trace d'erreur $I \longrightarrow t_1(i, j) \longrightarrow t_2(i) \longrightarrow \Theta$.

Il s'agit d'une trace fallacieuse car, pour exister, elle suppose que le programme contient *au moins* deux *threads*, ce qui aurait changé la première pré-image de Θ calculée par Cubicle. En effet, après avoir appliqué $t_1(i, j)$ à l'état $\mathtt{X}[i] = \mathsf{A} \wedge \mathtt{X}[j] = \mathsf{A}$, on obtient l'état $\mathtt{X}[i] = \mathsf{B} \wedge \mathtt{X}[j] = \mathsf{A}$ et la transition t_2 n'est plus applicable à cause de sa garde universelle.

Pour remédier à ce problème, on a modifié l'algorithme de Cubicle pour vérifier que les traces sont tout le temps possibles au moment où on effectue notre analyse d'atteignabilité arrière. Cette vérification met en œuvre une forme d'exploration en avant symbolique. Les résultats de nos expérimentations avec ce nouvel algorithme sont présentés dans la table ci-dessous.

	Avec raffinement				Sans raffinement			
	nœuds	Inv.	Restarts	Temps	nœuds	Inv.	Restarts	Temps
`sb_alt`	216	30	0	**1,70s**	215	152	7	7,64s
`sb.c` [15]	484	67	0	**3,26s**	331	226	10	20,7s
`sb_nice.c`	363	52	0	**2,06s**	240	106	9	11,6s
`sb_single.c`	192	27	0	**0,97s**	165	115	5	3,11s
`sb_single_us.c`☠	810	/	0	6,91s	810	/	0	**5,06s**
`sb_loop.c`	2146	257	0	**45,6s**	1274	1577	33	14m49s

7. Conclusion

Nous avons présenté dans cet article un outil pour vérifier la sûreté de barrières de synchronisation écrites en langage C pour un nombre quelconque de *threads*. Notre technique consiste à compiler un fragment du langage C avec *threads* vers le langage d'entrée du *model checker* Cubicle. Nous avons mené des expérimentations sur plusieurs implémentations de barrières *sense reversing*. Les résultats obtenus montrent la viabilité de l'approche.

D'autres approches existent pour la vérification de programmes C concurrents. On trouve par exemple des approches semi-manuelles comme la vérification déductive avec VCC [7] ou celles basées sur des assistants à la preuve comme Isabelle ou Coq (*e.g.* projet L4.verified [18]). En ce qui concerne les techniques automatiques, on trouve des approches par interprétation abstraite, comme DUET [11], qui s'attaquent à la vérification de propriétés simples (comme l'absence de déréférencement de pointeurs nuls) pour des programmes paramétrés. On trouve également des *model checkers* comme VeriSoft [13], CMC [23] ou CHESS [24] qui sont limités à la vérification de programmes C avec un nombre fixe (et petit) de *threads*. Parmi ces approches par *model checking*, on peut citer les travaux récents de Jiang et Jonsson [17] qui compilent un fragment du C vers le langage de Spin. Là encore, l'approche ne permet pas de prouver la sûreté pour un nombre quelconque de *threads*. Enfin on peut noter des techniques plus éloignées utilisant des systèmes de permissions pour prouver des modèles de barrières de synchronisation [20].

Ce travail constitue à nos yeux une approche prometteuse pour la vérification de programmes C concurrents. Pour poursuivre, nous envisageons d'étendre le modèle mémoire de notre compilateur pour y ajouter par exemple la notion de registre, de mémoire cache etc. Pour gagner en flexibilité, nous pensons qu'il serait intéressant de placer cette phase de vérification sur un langage intermédiaire d'un vrai compilateur C. Cela nous permettrait également de simplifier le schéma de traduction en vue de faire une preuve formelle de sa correction.

Références

[1] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *Verification, Model Checking, and Abstract Interpretation*, pages 22–36. Springer, 2008.

[2] S. V. Adve and H.-J. Boehm. Memory models : a case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8) :90–101, Aug. 2010.

[3] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *JSAT*, 8(1/2) :29–61, 2012.

[4] J. Alglave and L. Maranget. Stability in weak memory models. In *CAV*, pages 50–66, 2011.

[5] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1) :110–135, 1975.

[6] E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[7] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC : A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[8] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle : A Parallel SMT-based Model Checker for Parameterized Systems. In *CAV*, pages 718–724. Springer, 2012.

[9] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaidi. Invariants for Finite Instances and Beyond. In *FMCAD*, Portland, Oregon, USA, October 2013.

[10] S. Conchon, A. Mebsout, and F. Zaïdi. Vérification de systèmes paramétrés avec Cubicle. In *Vingt-quatrièmes Journées Francophones des Langages Applicatifs*, Aussois, France, Feb. 2013.

[11] A. Farzan and Z. Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In J. Field and M. Hicks, editors, *POPL*, pages 297–308. ACM, 2012.

[12] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving : Termination and invariant synthesis. *LMCS*, 6(4), 2010.

[13] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.

[14] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1) :1–17, 1988.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier, 2012.

[16] IEEE. *IEEE 1003.1c-1995 : Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2 : Threads Extension (C Language).* 1995.

[17] K. Jiang and B. Jonsson. Using spin to model check concurrent algorithms, using a translation from c to promela. In *In Second Swedish Workshop on Multi-Core Computing*, 2009.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4 : Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[19] L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 1979.

[20] D.-K. Le, W.-N. Chin, and Y.-M. Teo. Verfication of static and dynamic barrier synchronization using bounded permissions. In *ICFEM*, 2013.

[21] J. Lee and D. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*, 50 :824–833, 2001.

[22] N. A. Lynch. *Distributed algorithms.* Morgan Kaufmann, 1996.

[23] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC : A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI) :75–88, 2002.

[24] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 267–280. USENIX Association, 2008.

[25] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4) :319–340, 1976.

[26] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, volume 2031, pages 82–97. Springer, 2001.

[27] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

What could Coq do for Database Software? —A Progress Report

Yoichi Hirai & Reynald Affeldt

National Institute of Advanced Industrial Science and Technology, 1-1-1 Umezono, Tsukuba, Japan
{y-hirai,reynald.affeldt}@aist.go.jp

Abstract

We are storing more and more sensitive data on internet-accessed databases and, at the same time, database software is being optimized to accommodate larger data sets and an increasing number of concurrent accesses, to the point that the correctness of database software has become a subject of concern. We discuss three research directions and intermediate results towards the formal verification of database software using the Coq proof-assistant. The first on reversible printer-parsers aims at verifying the various interfaces of databases. In particular, we show that it helps mitigating the problem of command injection attacks. We verified several combinators for reversible printer-parsers and used them to build a safer printer-parser for SQL literals. The second direction is on database normalization. We have formalized and simplified a classic proof for computing the third normal form schema. The third direction is on index structures. We have formalized generalized search trees and verified the search operation for these trees. We believe the above experiments help pave the way towards the formal verification of database software.

1. Introduction

We are storing more and more sensible data, such as personal, medical or credit information on databases. These databases are often open to internet access and become targets of security attacks. At the same time, implementation and maintenance techniques are gaining in complexity. In addition to standard good practice such as database normalization, database software is also being optimized in various ways. The pursuit for performance often results in more complicated implementation. The caching mechanism together with write-ahead log mechanism aims at higher performance while keeping consistency of the database systems. Ideally, each operation should take the least possible number of locks (or any other synchronization method), but reasoning about such minimal sets of locks is error-prone. The importance of stored data and the complexity of recent implementations calls for strong measures to guarantee the security of transactions.

One convincing way to tackle this problem is to proceed to a detailed verification of the database software to, at least, assess its weaknesses and, at best, provide strong security guarantees. Formal verification has recently been investigated as an approach for this purpose. In [18], the authors formalized in the Coq proof-assistant an index-based implementation and verified it against an abstract model. But their work does not address the issue of database normalization and their formalization does not consider concurrency. Moreover, it is not clear how their formalization helps providing guarantees against security attacks such as command injection attacks. Another work related to formalization of databases is the formalization of Armstrong's laws in Mizar [21]. This work provides the premises for a theoretical study of relational models but does not go as far as formalizing the normal forms of schemas. Despite their limitations, these pieces of work show that formal verification of databases using proof-assistants is a promising approach. Yet, we believe that database software

encompasses so many different components (string processing, normalization, query optimization, indexing, etc.) that a monolithic approach to the problem is difficult. On the other hand, all the components of database software can also be useful taken separately. For this reason, our approach to the problem of verification of database software is to incrementally identify and verify modular components.

In this paper, we report on the formalization in Coq[1] of several pieces of database software:

- In order to solve encoding and decoding problems such as injection attacks, we have been developing reversible printer-parsers so that any abstract syntax tree is printed into a string which is parsed back to the identical tree. This allows, for example, the web application developers to focus on the SQL abstract syntax trees and forget about the SQL strings. This is the topic of Sect. 2.
- Database normalization is a well established theory that plays a key role in the efficient management of stored data. We have observed that classic proofs from the 1970's can be advantageously revisited using formal verification. For illustration, we pick up a pioneering paper on the third normal forms [6] and show that its non-trivial (an erroneous attempt had been made before [25]) proofs can be simplified by eliminating reasoning using graphical objects. This is the topic of Sect. 3.
- Several bugs have been found in the concurrent index trees of the widely used database management system PostgreSQL. Since concurrency makes testing ineffective in practice, it is important to work towards formal verification of concurrent index trees. As preliminary steps toward the concurrent case, we have formalized in Coq a memory model for GiST tree structures and verified a sequential search algorithm with it. This is the topic of Sect. 4.

We discuss related work in Sect. 5 and future work along with the conclusion in Sect. 6. An appendix provides complementary information about the specific topic of reversible computations (Appendix A).

2. Reversible Printer-Parsers for Database Systems

All interfaces of the database systems involve string printing and parsing. The server has to parse query strings and database system often come with a client side library that generates the query strings. For crash recovery, many database systems print write-ahead logs to disks, recording operations. After a crash, the logs must be parsed back correctly, but sometimes they are not. For example, PostgreSQL version 8.4.3 contains a bug fix on write-ahead log printing.

When computers print and parse strings, an important property is reversibility: the printed data is parsed back correctly. If we test printers and parsers for reversibility, it is hard to cover all relevant cases because a parser is essentially a complicated case analysis. Using Coq, we can verify that when the printer prints an abstract syntax tree into a string, then the parser parses the string into an abstract syntax tree identical to the original.

In this section, we first show how to compose simpler reversible printer-parsers into more complex ones while automatically keeping the formal proof of reversibility. We choose a modular strategy that allows us to reuse simple building blocks for different purposes: for example, a reversible printer-parser for escaped string literals can be used for SQL and for email addresses.

In Sect. 2.1, we introduce a formalization for reversible printer-parsers. In Sect. 2.2, we propose several generic combinators that we extend in Sect. 2.3 to form a reversible printer-parser for a subset of SQL. As an application, we further discuss prevention of command injection attacks in Sect. 2.4.

[1] Our development is in Coq 8.4pl2, SSReflect1.5rc1 and MathComp 1.5rc1, all found at http://coq.inria.fr/ or http://ssr.msr-inria.inria.fr/FTP/.

2.1. Reversible Parser-Printers

Before delving into the details of formalization, we discuss our design choices.

Types. At first sight, a printer for data of type `A` can take a term of type `A` and output a string; it may therefore have type `A → string`. As for the parser, since it can fail, it may have type `string → option A`. One problem with this choice of types is that the parser can only work after the whole string has arrived. Worse, in order to use this parser, we need a mechanism to transmit the endpoint of a string, which causes an additional encoding-decoding problem. To solve these problems, we allow the parser to consume some initial parts of the string and leave the rest. We change the type of parser to `string → option (A * string)`. Symmetrically, now the printer is of type `(A * string) → string`.

Preconditions. Unconditional reversibility does not hold for some frequently used pieces of syntax. We choose to attach printers' preconditions to reversible printer-parsers and certify reversibility assuming these preconditions. A precondition can be trivial, when any input satisfies it. Such unconditional reversibility indicates an unambiguous syntactic block that can be separated from whatever trailing string.

Specifications. In order to prove termination of some composite parsers (that perform for example repetition) and to compose conditional reversible printer-parsers, we need to annotate the parsers with Hoare-logic style specifications.

The reversible printer-parsers are defined as follows[2]:

```
Record syntax A spec precond :=
  { print : (A * string) → string;
    parse : string → option (A * string);
    reverse : ∀ a str, precond (a, str) → parse (print (a, str)) = Some (a, str);
    spec : ∀ str a rest, parse str = Some (a, rest) → spec str (a, rest) }.
```

The `reverse` proof establishes reversibility. The `spec` proof establishes the parser's specification: when the parser reads `str` of type `string` and successfully returns a pair `Some (a, rest)`, the input and the output satisfy the specification `spec str (a, rest)`.

2.2. General-Purpose Combinators for Reversible Printer-Parsers

One can easily define a reversible printer-parser for a single character at the leftmost tip of a string. Once we obtain the reversible printer-parser for one character, we can develop more complicated reversible printer-parsers by using the combinators below. These constructions are inspired from the invertible syntax descriptions [2, 20]. We mainly comment on the types of the combinators; their complete formalization is available online[3].

Alternative. When we have two reversible printer-parsers for data types `A` and `B`, we can build a printer-parser for the disjoint union `A + B`. The precondition and the specification of the lattar can be computed automatically from the preconditions and specifications of the elementary reversible printer-parsers. Here, we need to address a confusion problem: a term of type `B` could be printed and then parsed back to a term of type `A`. Our solution is to explicitly require in the precondition that the input `b` of type `B` does not cause such a confusion as observed in the underlined conjunct below:

```
Let Pab := [fun c : (A + B) * string ⇒ match c.1 with
            | inl a ⇒ Pa (a, c.2)
            | inr b ⇒ Pb (b, c.2) ∧ parse elmA (print elmB (b, c.2)) = None end].
Let Sab str := [fun c : (A + B) * string ⇒ match c.1 with
```

[2] This is actually an instance of a general reversible computation that we expand for the sake of clarity. See Appendix A for the generic properties of reversible computations.

[3] http://staff.aist.go.jp/y-hirai/db.html

```
                    | inl a2 ⇒ Sa str (a2, c.2)
                    | inr b1 ⇒ Sb str (b1, c.2) ∧ parse elmA a = None end].
Definition rev_alt : syntax (A + B) Sab Pab.
```

elmA and elmB above are the elementary reversible printer-parsers for A and B respectively. When the input contains a data of type B, the precondition Pab requires the underlined non-confusion property in addition to the original precondition Pb. When it is A, the precondition requires nothing more than the original precondition Pa. This is safe because the alternative parsing first tries to parse the string as A and then tries to parse the string as B only if the first attempt fails. The definition uses more general combinators for general reversible computation (see Appendix A).

Pairing. The pairing printer-parser consecutively uses two reversible printer-parsers. Let us consider a reversible printer-parser with precondition Pa : A * string → Prop and another printer-parser with precondition Pb : B * string → Prop and specification Sb. We can define the type of the pairing reversible printer-parser as follows:

```
Let Pab (w : A * B * string) := (∀ str, Sb str (w.1.2, w.2) → Pa (w.1.1, str)) ∧ Pb (w.1.2, w.2).
Let Sab (r_str : string) (w : A * B * string) := ∃ str, Sa r_str (w.1.1, str) ∧ Sb str (w.1.2, w.2).
Definition pairing : syntax (A * B) Sab Pab.
```

To understand intuitively the precondition Pab, one can think of the pairing printer as first printing B and then printing A (from right to left). The precondition Pab establishes two preconditions for the two printers. One might wonder why the precondition Pab contains a specification Sb, which is about the parser. In fact the specification Sb also applies to the printer because the printer's input and output must be, respectively, the parser's output and input, which must obey the specification Sb. As for the specification of the pairing, it is just the relational composition of the specifications.

Repetition. A reversible printer-parser can be used repeatedly: one can try to use the same parser as many times as successfully executed. From a reversible printer-parser for data type A, the many operator makes a printer-parser for data type seq A. The repetition printer-parser has a precondition stating that the original element printer-parser cannot parse the trailing string. Also, the original printer-parser must consume at least one character on each successful parsing.

Data conversion. Given a reversible printer-parser for data type B and a reversible conversion between C and B, we can combine them and obtain a reversible printer-parser for data type C. Such reversible conversions are discussed in Appendix A.

Consequence. Named after the consequence rule of the Hoare logic, this combinator can weaken the specification and strengthen the precondition of a reversible printer-parser.

2.3. Special Printer-Parsers for SQL

On top of the above general purpose combinators, we further developed printer-parsers for SQL components. As a target, we choose EXECUTE sentences of SQL, which covers a large portion of possible database operations.

Literals. There are three kinds of literals: string literals, numeric literals and some special identifiers (including NULL). The most complicated class of literals turns out to be the numeric literals, whose syntax is illustrated in Fig. 1. Our printer-parser for the numeric literals has a precondition on the trailing string; the trailing string must not start with any numeric character or upper 'E' character. Our choice of precondition is not the weakest possible, but it is weak enough in the context of EXECUTE sentences.

EXECUTE sentences. For a whole EXECUTE sentence, we obtained a reversible printer-parser with unconditional reversibility. Below is an example with two literals and a trailing string (underlines show corresponding segments):

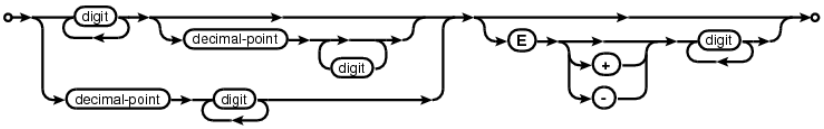


Figure 1: The syntax of a numeric literal, taken from http://www.sqlite.org/lang_expr.html.

```
Eval vm_compute in parse execute_statement "EXECUTE func 'literal' 32.5E4 ; rest".
= Some (inl char_f, [:: inl (inl char_u); inl (inl char_n); inl (inl char_c)],
[:: inl (inr "literal"); inl (inl (Usual (three, [:: two], Some [:: five]), Some (None, (four, [::])))))], " rest")
```

(The list notation [:: a; b; c] is in the SSReflect library.) The parsed string contains an EXECUTE sentence followed by a trailing string " rest".

2.4. Application: Prevention of Command Injection Attacks

Command injection attacks exploit the possibility that user-inputs of a target system alter the syntactic structure of SQL command strings. We propose a method to prevent injection attacks using a formally verified reversible printer-parser. This method reduces the multi-party problem of command injection prevention to a single-party problem of abstract syntax tree construction.

Let us consider the following example (borrowed from [22]) of a command injection attack.

```
WHERE uname = "John" AND cardtype = 2 OR 1 = 1
```

is created by a database client (e.g., a web application) by substituting ? in the following template:

```
WHERE uname = "John" AND cardtype = ?
```

with the expression `2 OR 1 = 1` (which is a user-input string from the attacker). Since the substitute `2 OR 1 = 1` evaluates to 1 (in SQLite version 3.7.13 and 3.8.1, at least), one may think that the whole WHERE clause chooses the records whose uname is John and cardtype is 1. In fact, the database server interprets the same string as a different syntax tree, as parentheses show below:

```
WHERE ((uname = "John") AND (cardtype = 2)) OR (1 = 1)
```

The condition is a disjunction with the disjunct `1 = 1`, which always evaluates to true, so that the WHERE clause chooses all the records. This example illustrates that the cause of injection attacks is a mismatch between the two abstract syntax trees of the client and the server.

Reversible printing-parsing to prevent injection attacks. We propose to prevent mismatches responsible for injection attacks by using verified reversible printer-parsers for SQL. Instead of performing a dangerous substitution, the database client just has to construct an abstract syntax tree and let our reversible printer-parser make an SQL string, which is guaranteed to be parsed into the tree identical to the client-side abstract syntax tree.

It is important to handle operators with priorities, e.g., AND connects stronger than OR. Because our library does so, the following two parsing examples result in different abstract syntax trees:

```
Definition sans_parens := get third_term "UNAME = 'John' AND CARDTYPE = 2 OR 1 = 1".
Definition avec_parens := get third_term "UNAME = 'John' AND CARDTYPE = (2 OR 1 = 1)".
Goal ~ sans_parens = avec_parens. by vm_compute. Qed.
```

In the first example without parentheses, AND connects stronger than OR so that the OR node becomes the parent of the AND node. This is the contrary in the second example (see also Fig. 2). In this example,

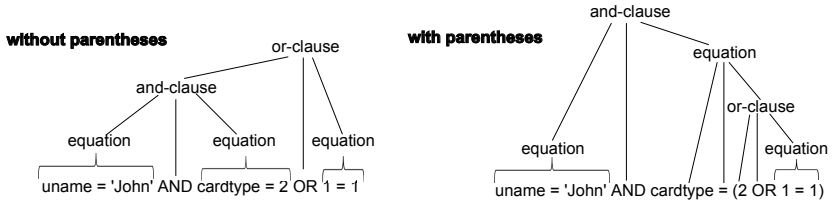


Figure 2: Parse trees for two similar SQL expressions with parentheses and without parentheses.

the reversible printer-parser has the name `third_term`, the name indicates that the printer-parser only allows three-depth parentheses. We are currently working on the generalization to arbitrary depths by using the length of parsed strings to calculate an upper bound for the depth.

3. Database Normalization

The normalization of database schemas (formal definitions follow soon afterwards) is a well-established part of the database theory (whose foundations were laid in the seventies). It nevertheless features non-trivial proofs. For example, Wang and Wedekind [25] provided in 1975 a wrong proof that their algorithm computes the third normal. For this reason alone, there is value in formalizing the database normalization. In addition, we show that classic proofs can be advantageously revisited using the modern technology of proof-assistants. Let us consider the paper by Bernstein [6] who, following Wang and Wedekind, proposed two algorithms for deriving schemas in the third normal form from any set of functional dependencies. We realized, for example, that Bernstein resorts to graph terminology to reason about functional dependencies where an inductive reasoning is sufficient.

Below, we first provide background on schemas and normal forms, we then explain a detour in the proofs of Bernstein (Sect. 3.1), and finally move on to our formalization of functional dependencies (Sect. 3.2) and of Bernstein's proof (Sect. 3.3).

Background on schemas and normal forms. A database *schema* is a finite set of table names (called *relations*), each associated with some column names (called *attributes*). The normal forms of schemas are defined with respect to *functional dependencies* (FDs) between attributes. An FD captures the situation where a set of attributes uniquely determines another attribute (for example, two attributes `flight_number` and `date` uniquely determine `departure_time`). The FD $X \triangleright Y$ is formally a pair of two (finite) sets X and Y of attributes.

The *third normal form* excludes database tables with the undesirable transitive dependency: $X \triangleright Y$, $Y \not\triangleright X$ and $Y \triangleright Z$. For example, suppose a table contains attributes `user_id`, `affiliation_id` and `affiliation_address`. Since `user_id` determines `affiliation_id` and in turn `affiliation_id` determines `affiliation_address`, the same address can be written at several places in the table; when the user tries to update an `affiliation_address`, he or she can easily forget to update some occurrences and break the expected functional dependency. Database designers enforce the third normal form to avoid this kind of anomalies (in the example above, by splitting the table in two).

3.1. A Detour in Bernstein's Proofs

Bernstein proposed and proved correct two algorithms for generating third normal form database schemas [6]. The proofs contain graph theoretical terminology ("leaf nodes", "the children of the node", etc.) on a structure as in Figure 3. More precisely, graph theoretical reasoning is used during the following step, especially the underlined phrases:

> Suppose there is a derivation for $X \triangleright A_i$ in H that uses $Z \triangleright A_i$. Then by Lemma 1, we have $X \triangleright Z$. But this violates $X \not\triangleright Z$ in the transitive dependency. So $X \triangleright A_i$ must be derivable without the use of $Z \triangleright A_i$.

The proof step above uses the following lemma:

> **Lemma 1.** Let G be a set of FDs, and let $g\colon X \triangleright Y$ be an FD in G. If $h\colon V \triangleright W$ is in G^+ and g is used for some derivation of h from G, then $V \triangleright X$ is in G^+.

Graph theoretical reasoning stems from the underlined parts that express the existence of derivations using a special assumption. In the next section, we provide a Coq formalization of the closure operation on FDs, replacing graph theoretical reasoning with simpler, inductive arguments.

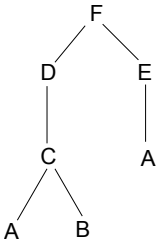


Figure 3: A sample derivation tree from [6], showing that $\{A, B\} \triangleright \{F\}$ is in the closure of $\{\{A, B\} \triangleright \{C\}, \{C\} \triangleright \{D\}, \{D, E\} \triangleright \{F\}, \{A\} \triangleright \{E\}\}$.

3.2. Closure Computation of Functional Dependencies in Coq

We first define the type of FDs. For this purpose, we assume a type `att : eqType` of attributes. A *simple dependency* is an FD whose right-hand side is a singleton:

```
Definition simple_dep := seq att * att.
```

We can restrict ourselves to simple dependencies because any FD can be reduced to a finite set of simple dependencies.

We now define the closure operation on the set of FDs. Bernstein's paper deals with the closure under the rules known as Armstrong's laws [3]:

Reflexivity If $Y \subset X$ then $X \triangleright Y$.
Augmentation If $Z \subset W$ and $X \triangleright Y$ then $X \cup W \triangleright Y \cup Z$.
Transitivity If $X \triangleright Y$ and $Y \triangleright Z$ then $X \triangleright Z$.

We formalize FDs as the following inductive predicate:

```
Section Theory.
Variable th : seq simple_dep.
Let theory fd := fd ∈ th.

Inductive closure : seq att → seq att → Prop :=
| orig: ∀ X A, theory (X, A) → closure X [:: A]
| sets: ∀ X X' Y Y', X =i X' → Y =i Y' → closure X Y → closure X' Y'
| fd_rfl:  ∀ (X Y : seq att), {subset Y ⊆ X} → closure X Y
| fd_aug:  ∀ X Y Z W, {subset Z ⊆ W} → closure X Y → closure (X ∪ W) (Y ∪ Z)
| fd_trans:  ∀ X Y Z, closure X Y → closure Y Z → closure X Z .
```

`closure X Y` represents the fact that the functional dependency $X \triangleright Y$ is in the closure. One can recognize Armstrong's laws: reflexivity is implemented by the constructor `fd_refl`, augmentation by `fd_aug`, and transitivity by `fd_trans`. Other constructors are for bootstrapping (`orig`) and to formalize the semantics of sets using the list library of SSReflect. We use the permutation equality `=i` on lists to simulate the set equality on finite sets.

Given a set of attributes X, we can compute the largest Y where $X \triangleright Y$ is in the closure of `th`. The algorithm is a functional version of a program from Maier's book [17, Algorithm 4.2 in Sect. 4.6].

1. (Eliminate extraneous [irrelevant, authors' note] attributes.) Let F be the given set of FDs. Eliminate extraneous attributes from the left side of each FD in F, producing the set G. An attribute is extraneous if its elimination does not alter the closure of the set of FDs.
2. (Find covering.) Find a nonredundant covering H of G.
3. (Partition.) Partition H into groups such that all of the FDs in each group have identical left sides.
4. (Construct relations.) For each group, construct a relation consisting of all the attributes appearing in that group. Each set of attributes that appears on the left side of any FD in the group is a key of the relation. (Step 1 guarantees that no such set contains any extra attributes.) All keys found by this algorithm will be called synthesized. The set of constructed relations constitutes a schema for the given set of FDs.

Figure 4: Bernstein's first algorithm for deriving the third normal form schema [6, Algorithm 1(a)].

It repeatedly adds elements to a set until saturation. In Coq where functions need to be provably terminating, we use the number of attributes involved as an upper-bound for the number of iterations:

```
Definition comp_closure_of : seq att → seq att := comp_closure_of_ X (num_attributes X).
```

This computation is sound and complete with respect to the definition of closure:

```
Lemma comp_closure_of_sound : ∀ X, closure X (comp_closure_of X).
Lemma comp_closure_of_complete : ∀ X Y, closure X Y → {subset Y ⊆ comp_closure_of X}.

Definition comp_closure X Y := inc Y (comp_closure_of X).
End Theory.
```

We introduce the following notations for closures of a theory `th : seq simple_dep`:

```
Notation " ( th ')' '⊢' X ▷ Y" := (closure th X Y).
Notation " [ th ']*' '∋' X ▷ Y" := (comp_closure th X Y).
```

3.3. Formal Verification of the Third Normal Form Computation

We verified the first algorithm of Bernstein, displayed in Fig. 4, for computing third normal form schemas. More precisely, we have completed the proofs that the steps 1.–4. indeed produce a third normal form schema:

```
Theorem algorithm1a_3NF : ∀ fds key sch, (key, sch) ∈ (algorithm1a fds) → NF3 fds sch.
```

This lemma gives us formal grounds to claim that Bernstein really fixed the proof by Wang and Wedekind. We now explain in details the formalization of `algorithm1a` and `NF3`.

Formalization of the algorithm. The algorithm lends itself well to formalization. The steps are translated into three function applications:

```
Definition algorithm1a fds := map relationify (classify (find_covering (remove_extra fds))).
```

For illustration, let us comment on the formalization of step 2. The purpose of step 2 is to find a minimal subset of the input without changing the closure. It is implemented by repeated application of the function `find_covering_step`. Termination is guaranteed by the decreasing sizes of the subsets:

```
Function find_covering fds {measure size fds} :=
 match find_covering_step fds with | (fds', true) ⇒ fds' | (fds', false) ⇒ find_covering fds'  end.
```

Properties of `find_covering_step` can be turned into properties of `find_covering` using the induction scheme `find_covering_ind` generated by `Function`. We use it for example to prove that the closure is left unchanged. The second lemma below can be proved by the first, using `find_covering_ind`:

```
Lemma find_covering_step_spec : ∀ fds, equiv fds (find_covering_step fds).1.
Lemma find_covering_keeps_sem : ∀ fds, equiv fds (find_covering fds).
```

Formalization of the third normal form. The formalization of the definition of the third normal form is a bit technical. Formally, a relation is in the third normal form "if none of its nonprime attributes are transitively dependent upon any key" [6]:

```
Definition NF3 (r : relation) := ∀ A K, key K r → nonprime A r → ~ transitive_dependent r A K.
```

An attribute A_i is *transitively dependent* upon a set of attributes X "if there exists a set of attributes $Y \subseteq \{A_1, \ldots, A_n\}$ such that $X \triangleright Y, Y \not\triangleright X$ and $Y \triangleright A_i$ with A_i not an element of X or Y" [6]. We assigned a single variable `r` to the relation $\{A_1, \ldots, A_n\}$.

```
Definition transitive_dependent th (r : relation) (Ai : att) X :=
  ∃ Y, {subset Y ⊆ r} ∧  Ai ∉ X ∧ Ai ∉ Y ∧ (th) ⊢ X ▷ Y ∧ ~ (th) ⊢ Y ▷ X ∧ (th) ⊢ Y ▷ [:: Ai] ∧ Ai ∈ r.
```

We now explain and formalize key-related technical terms. A set of attributes of a relation is a *superkey* when it uniquely determines all attributes in the relation. A minimal superkey is called a *key*:

```
Definition superkey th (X : seq att) (r : relation) := {subset X ⊆ r} ∧ (th) ⊢ X ▷ r.
Definition key th (X : seq att) (r : relation) :=
  superkey th X r ∧ ∀ X', {subset X' ⊆ X} → superkey th X' r → X =i X'.
```

Finally, a *nonprime* attribute is not contained in any key:

```
Definition nonprime th (A : att) (r : relation) := A ∈ r ∧ ∀ K, key th K r → A ∉ K.
```

Proof excerpt. We now explain how we carry out in Coq the pencil-and-paper reasoning step by Bernstein presented in Sect. 3.1. In the course of the formal proof, we face the following goal where we want to prove that $X \triangleright A$ is derivable without $Z \triangleright A$:

```
  ...
  XZnot : ~ (fds) ⊢ X ▷ Z
  XA : (fds) ⊢ X ▷ [:: A]
  H := find_covering fds : seq simple_dependency
  ============================
   (remove' (Z, A) H) ⊢ X ▷ [:: A]
```

This is a proof step by contradiction. Let us assume ab absurdo that every derivation $X \triangleright A$ use $Z \triangleright A$. Then we can use (a variant of) Lemma 1 (see Sect. 3.1) to prove that $X \triangleright Z$ is in the closure:

```
  XA : (fds) ⊢ X ▷ [:: A]
  H := find_covering fds : seq simple_dependency
  I : ~~ [remove' (Z, A) H]* ∋ X ▷ [:: A]
  ============================
   (H) ⊢ X ▷ Z
```

This contradicts the assumption `XZnot : ~ closure fds X Z`.

The variant of Lemma 1 that we used improved Berstein's lemma by being free of any specific derivations; we only have universal quantifications:

```
Lemma l1' : ∀ G X y V W, ([G]* ∋ V ▷ W) → ~~ ([remove' (X, y) G]* ∋ V ▷ W) → ([G] ⊢ V ▷ X).
```

This property only depends on the closures and not on the form of derivation trees shown in Fig. 3. In comparison, the original version of Lemma 1 assumed existence of such a derivation, which is why Bernstein took a detour to analyze the structure of derivations.

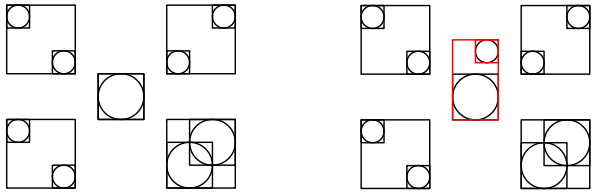


Figure 5: An index before (left) and after (right) an insert operation. There, index keys are rectangles and data are circles. When a circle is inserted, one of the rectangles is expanded to contain it. The rectangle is chosen so that the expansion is the smallest. These pictures are drawn in Coq by `vm_compute in (draw_ms ms n0)`, which produces a script for Asymptote (http://asymptote.sourceforge.net).

4. Generalized Index Structures

Indexes are the component of database systems that makes the retrieval of information faster. The best performance is obtained using concurrent indexes that threatens the correctness of the implementation. For illustration, one may find bug fixes on GiST indexes in release versions for PostgreSQL (versions 8.4.3 and 8.4.17). Because of the interleaving (or worse, non-serializable) execution of several processes, concurrent indexes have a highly non-deterministic behavior that makes testing ineffective. Consequently, formalization using a proof-assistant such as Coq appears as the only way to guarantee a correct implementation.

The most widely used index data structures are variants of B+trees [16], which are generalized by "generalized search trees" (GiST) [14]. GiSTs are implemented in PostgreSQL and used in particular for geographical data (PostGIS) and genome sequences (BioPostgres). GiSTs generalize the types of the searched data and the search keys. Unlike B+trees, the keys do not have to be totally ordered. For example, the searched data can be a circle in the plane, the key can be a rectangle, and one can search the index with a rectangle for all the circles contained in the rectangle (see Fig. 5). The user of GiST must supply the types of data and keys along with several functions satisfying a fixed set of axioms. The algorithm uses the user-supplied functions for choosing which branch to follow. The user-supplied functions must satisfy some axioms, which are written on paper [14] but only used in informal descriptions.

In this section, we investigate the formalization of GiST searching algorithms. As a first step, we restrict ourselves to the sequential case. In Sect. 4.1, we first formalize a simple memory model. In Sect. 4.2, we verify a GiST searching algorithm using this model.

4.1. The Memory Model for GiST

A GiST is a tree with a single top node and many leaf nodes. A node contains a sequence of entries and each entry contains a pointer to a child node or a piece of data. The users of GiST must specify a set of predicates on data. Since the predicates are stored in memory and are also used as search queries, we assume a type `pred : eqType` for predicates accompanied by an interpretation function:

```
Variable pred : eqType.
Variable pred_interp : pred → data → bool.
```

The memory model supporting the tree structure is formalized as follows. We assume a type `pointer : eqType`. An entry is a pair of a predicate and a pointer:

```
Inductive entry : Set := Entry : pred → pointer → entry.
```

One can look at the predicate to decide whether or not to follow the pointer. A pointer state is either empty, a node or a piece of data, and a memory state is just a function from pointers to pointer states:

```
Inductive ptr_state : Set := | Empty : ptr_state | Node  : node → ptr_state | Data  : data → ptr_state.
Definition memory_state := pointer → ptr_state.
```

A node consists of a level and entries: `Record node : Set := mkNode { level : nat; content : seq entry }.`

Pointers can be explored for nodes or data. For example, given a memory state `ms` and a pointer `ptr`, one can try to take data in a pointer state, or similarly a node, using the `pick_node` function.

```
Definition pick_data e := match e with Entry p ptr ⇒ (if ms ptr is Data d then Some d else None) end.
```

4.2. A Verified GiST Searching Algorithm

The GiST searching algorithm uses a user-supplied function called `consistent` that checks an entry against a query (represented by a predicate). When `consistent` returns false, following the pointer in the entry should yield no results so that the searching algorithm can skip the entry. For this optimization, we need a one-way soundness for the case where the function returns false:

```
Variable consistent: entry → pred → bool.
Variable consistent_sem: ∀ (p : pred) (ptr : pointer) (q : pred),
  ~~ (consistent (Entry p ptr) q) → ∀ x: data, pred_interp p x →  ~~pred_interp q x.
```

Also the tree must maintain a property: the data reachable below an entry `Entry p ptr` must satisfy `pred_interp p`. This constraint is contained in the `valid` predicate.

The GiST searching algorithm below can start from any level `from_level` and any entry `from_e`. The search goes as deep as `to_level`, which is set to 0 when the function is used to find some leaf entries. It uses `consistent` as follows:

```
Fixpoint search (ms : memory_state) (q : pred) from_level (from_e : entry) to_level : seq entry :=
 if  ~~(consistent from_e q) || (from_level < to_level) then [::]
    (* query and entry are not consistent or levels invalid, so no entries *)
 else if from_level == to_level then [:: from_e]  (* already reached deep enough, search finishes *)
      else match from_level with (* analyze the current level *)
        | 0 ⇒ [::]        (* from a leaf entry, no entry reachable *)
        | next_level.+1 ⇒
          oapp (fun n ⇒ flatten (map (fun next ⇒ search ms q next_level next to_level) (content n)))
              [::] (pick_node ms from_e) (* examine the entry to find the child entries, recur *) end.

Definition search_data ms q from_level from_e := pmap (pick_data ms) (search ms q from_level from_e 0).
```

Here, `pmap` and `oapp` are the map operation and the functional application for partial functions.

We verify that the algorithm above is correct in the sense that it does not skip any legit result. The idea of the statement is to compare the result of the search with a straightforward implementation that scans all the elements:

```
Lemma contained_found : ∀ ms (level : nat) (e : entry) (q : pred), valid ms level e →
filter (pred_interp q) (reachable_data ms level e) = filter (pred_interp q) (search_data ms q level e).
```

The comparison is between the above described search operation and the reachable function. Their definitions are different in the presence and the absence of the `consistent` function. While the search operation ignores entries not consistent with the query, the reachable function follows these entries as well. The lemma states that, the search operation does not miss any data by ignoring the inconsistent entries. Naturally, the lemma uses the hypothesis `consistent_sem`, and the assumption `valid` contains the assumptions on the shape of the tree as mentioned above.

We have also implemented (but not proved yet) the insertion algorithm. It is illustrated by the pictures in Fig. 5. We are in the process of verifying the insertion against the multiset semantics.

5. Related Work

About reversible printer-parsers. Trailing strings come from the parser libraries in Haskell, but we modified the types for the sake of reversibility (see Sect. 2.1). For example, Hutton [15] used lists to express nondeterministic outcomes of parsers. This is of course not compatible with reversibility: the parser must be deterministic in order to parse back the originally printed data uniquely.

The Hoare-style precondition and specification seen in Sect. 2.1 come from the Hoare state monad [23]. Uezato [24] already used this monad for showing termination of parser combinators. We further exploit this idea to enable modular composition of reversible printer-parsers.

There are several methods [19, 7, 11] for producing pairs of a parser and a pretty-printer. Invertibility is stated only conditionally if ever. Rendel and Ostermann [20] developed invertible syntax descriptions as a Haskell library of printer-parsers combinators. Reversibility is not guaranteed because the printers may not be total and the union operator $\diamond$ does not preserve reversibility when its arguments are not disjoint. For such operations, we defined how to compute a precondition automatically so that reversibility holds. Danielsson [9] used Agda to verify pretty-printers; however, reversibility is only guaranteed when the grammar is unambiguous.

Affeldt, Nowak and Oiwa [2] verified a fragment of invertible syntax descriptions for network binary packets. They introduce a new dependent construction (that chooses the next parsers depending on the data previously parsed). Our work improves their work by providing a verified choice operator, which was essential in implementing reversible printer-parsers for inductive data structures.

Reversible computation [4, 5] influenced our design of the reversible printer-parsers and is the reason for the generalization we provide in Appendix A.

Formalization of command injection attacks. Su and Wassermann [22] defined command injection attacks solely on user-input strings and abstract syntax trees. Namely, an SQL query is not a command injection attack if every piece of user-generated strings in the query is a valid syntactic form, i.e., contained in a specified kind of subtree like literals in the parse tree. Thus in order to prevent command injection attacks, we only have to prepare a correct abstract syntax tree on the client and transmit the identical tree to the server. The application of reversible printer-parsers we presented in Sect. 2.4 deals with the transmission part and reduces the problem of command injection prevention to a simpler problem of preparing an abstract syntax tree on the client side.

Su and Wassermann [22] also developed an SQL injection detector that treats a web application as a blackbox. At the execution time, the detector injects a special form of pseudo-parentheses strings around the user-inputs and searches the produced SQL queries for the pseudo-parentheses strings. Their approach can only deal with web applications that paste user-inputs into SQL queries.

Formalization of database theory. Armstrong, Nakamura and Rudnicki [3] formalized Armstrong's laws in Mizar [21], along with the proof that the laws are sound and complete with respect to the functional dependencies of database schemas. In addition we verified closure computation based on the Armstrong's laws and furthermore applied it to the computation of third normal forms.

Malecha, Morrisett, Shinnar and Wisnesky [18] verified the sequential execution of the B+tree against a relational data model in Coq but did not address normalization or SQL printing-parsing. The Ynot team's verification only covered sequential executions.

Verifying concurrent data structures. The planned verification of concurrent GiST would belong to the large field of verification of concurrent data structures where automatic verification tools are actively developed. Abdulla et al. [1] developed an automatic verification tool which is capable of handling unlimited number of threads and infinite data domains. Da Rocha Pinto et al. [8] verified the concurrent version of B+tree using a technique called concurrent abstract predicates [10]. The formal verification of the above tools have yet to be addressed.

6. Conclusion

In this paper, we discussed formalization of three core components of database systems. In Sect. 2, we formalized generic combinators for reversible printing-parsing, applied them to the particular case of SQL queries, and showed that the formal guarantee of reversibility improves security by mitigating the risk of command injection attacks. In Sect. 3, we verified the third normalization of database schemas. As exemplified by the work of Wang and Wedekind [25], correctness proofs for normalization algorithms are non-trivial. Using Coq, we were able to improve the classic proof of correctness for the third normal form algorithm of Bernstein. In Sect. 4, we investigated the formalization of index structures. We illustrated the issue with a memory model and the formalization of a searching algorithm.

Future work. We plan to verify Bernstein's second algorithm for producing third normal form schemas. This second algorithm improves the first one by using the smallest possible number of relations. Both algorithms share many steps, so that we should be able to reuse several lemmas. Other candidates for formalization are of course the fourth [12], the fifth [13] and the Boyce-Codd [17] normal forms. This work could lead to a verified database schema designing tool, and provide a formal relational model that could be used to verify the correctness of query optimization.

We plan to tackle formalization of concurrent index structures using linearizability. Linearizability is a standard for concurrent algorithms that requires any concurrent execution to simulate successive atomic executions. For practical reasons, we will start from the sequential version of GiST using the memory model presented in this paper and extend the formalization into a concurrent version.

Our current formalization of the reversible printer-parser uses disjunctive sums. Consequently, many parsing results contain non-informative injections (`inl`'s and `inr`'s). We can already use data conversion to convert each inductive structure into simple disjunctive sums, but it would be more practical to find a definition of reversible printer-parsers for general inductive structures. Our formalization of reversible printing-parsing may benefit from several improvements to ease the development of a complete SQL printer-parser (e.g., by allowing left-recursive grammars).

Acknowledgments. We thank the anonymous reviewers for the detailed comments and additional pointers to relevant work. The first author acknowledges partial support from JSPS Grants-in-Aid for Scientific Research 25880032. We used Mahboubi's `lstcoq.sty` to typeset code. Some fontshapes come from a literate Haskell code by Rendel and Ostermann [20].

References

[1] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *Proc. of the 19th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 324–338. Springer-Verlag, 2013.

[2] R. Affeldt, D. Nowak, Y. Oiwa. Formal network packet processing with minimal fuss: invertible syntax descriptions at work. In *Proc. of the 6th Wksp on Programming Languages meets Program Verification*, pages 27–36. ACM, 2012.

[3] W. Armstrong, Y. Nakamura, P. Rudnicki. Armstrong's axioms. *Formalized Mathematics*, 11(1):39–51, 2003.

[4] H. B. Axelsen, R. Glück. A simple and efficient universal reversible Turing machine. In *Proc. of the 5th Int'l Conf. on Language and Automata Theory and Applications*, volume 6638 of *Lecture Notes in Computer Science*, pages 117–128. Springer, 2011.

[5] C. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17:525–532, 1973.

[6] P. A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Trans. Database Syst.*, 1(4):277–298, 1976.

[7] R. J. Boulton. Syn: a single language for specifiying abstract syntax tress, lexical analysis, parsing and pretty-printing. Tech. Report UCAM-CL-TR-390, University of Cambridge, Computer Laboratory, 1996.

[8] P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, M. Wheelhouse. A simple abstraction for complex concurrent indexes. In *Proc. of the 2011 ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications*, pages 845–864. ACM, 2011.

[9] N. A. Danielsson. Correct-by-construction pretty-printing. In *Dependently-Typed Programming 2013*, to appear, 2013.

[10] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, V. Vafeiadis. Concurrent abstract predicates. In *Proc. of ECOOP*, pages 504–528. Springer-Verlag, 2010.

[11] J. Duregård, P. Jansson. Embedded parser generators. In *Proc. of Haskell 2011*, pages 107–117. ACM, 2011.

[12] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst.*, 2(3):262–278, 1977.

[13] R. Fagin. Normal forms and relational database operators. In *Proc. of SIGMOD 1979*, pages 153–160. ACM, 1979.

[14] J. M. Hellerstein, J. F. Naughton, A. Pfeffer. Generalized search trees for database systems. In *Proc. of 21th Int'l Conf. on Very Large Data Bases*, pages 562–573. Morgan Kaufmann, 1995.

[15] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[16] P. L. Lehman, S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[17] D. Maier. *Theory of Relational Databases*. Computer Science Press, 1983.

[18] G. Malecha, G. Morrisett, A. Shinnar, R. Wisnesky. Towards a verified relational database management system. In *Proc. of POPL 2010*. ACM, 2010.

[19] K. Matsuda, M. Wang. FliPpr: a prettier invertible printing system. In *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2013.

[20] T. Rendel, K. Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *Proc. of Haskell 2010*, pages 1–12. ACM, 2010.

[21] P. Rudnicki. An overview of the Mizar project. In *Proc. of the 1992 Wksp on Types for Proofs and Programs*, pages 311–330, 1992.

[22] Z. Su, G. Wasermann. The essence of command injection attacks in web applications. In *Proc. of POPL 2006*, pages 372–382. ACM, 2006.

[23] W. Swierstra. A Hoare logic for the state monad. In *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2009.

[24] Y. Uezato. Implementing monadic total parser combinator on Coq. *Information Processing Society of Japan Transactions on Programming*, 5(2):1–15, 2012. In Japanese.

[25] C. P. Wang, H. H. Wedekind. Segment synthesis in logical data base design. *IBM Journal of Research and Development*, 19(1):71–77, 1975.

A. A Generic Library for Reversible Computation

The reversible printer-parsers that we introduced in Sect. 2.1 are an instance of a more general library about pieces of reversible computation, which we call the *reversible elements*:

```
Record reversible (A B : Type) P Q :=
  { put : B → A;          (* forward function *)
    get : A → option B; (* backward partial function *)
    reverse : ∀ b, P b → get (put b) = Some b;
    spec : ∀ a b, get a = Some b → Q a b }.
```

This code block defines a reversible element with `put` and `get`. In addition to `put` and `get`, a term of type `reversible A B P Q` contains two proofs, namely, `reverse` for the reversibility and `spec` for a specification. The specification `Q` on the backward partial function is used in Sect. 2.2 to compose printer-parsers.

Composition. Suppose there are two reversible elements `AB : reversible A B bP abP` and `BC : reversible B C cP bcP`. We can compose their forward functions and backward partial functions respectively so as to define the composite reversible element of the following type:

```
rev_comp : reversible A C cP' acP.
```

The precondition `cP'` and the specification `acP` come from Swierstra [23].

Choice. The choice combinator is used to formalize the alternative combinator of Sect. 2.2. Suppose there are two reversible elements `CA : reversible C A aP caP` and `CB : reversible C B bP cbP`. These reversible elements encode values of types `A` and `B` into a common type `C`. We call the resulting reversible element the *choice* reversible element. For reversibility, it is enough to specify an input of `B` not to cause confusion, i.e., when printed and parsed back, misinterpreted as an element of `A`. Thus the precondition for the choice reverse element is:

```
Let abP (ab : A + B) := match ab with | inl a ⇒ aP a | inr b ⇒ bP b ∧ get CA (put CB b) = None end.
```

The specification of the choice reverse element is:

```
Let cabP c (ab : A + B) := match ab with | inl a ⇒ caP c a | inr b ⇒ cbP c b ∧ get CA c = None end.
```

Using these, the type of the choice reverse element is `choice : reversible C (A+B) abP cabP`.

Rendel's library[4] calls this operation `(|||)`. In the source code, he comments "this is not a proper partial isomorphism,"[5] apparently for the lack of invertibility. We overcome this difficulty by having preconditions and specifications attached to the reversible elements.

[4] A software library called `partial-isomorphisms`, whose source code is available at http://hackage.haskell.org/packages/archive/partial-isomorphisms/0.2/.

[5] The comment can be found in `src/Control/Isomorphism/Partial/Prim.hs`

Exécution efficace de programmes ReactiveML

Louis Mandel[1,3] & Cédric Pasteur[2,3]

1: Collège de France
2: École normale supérieure
3: INRIA Paris-Rocquencourt

Résumé

ReactiveML est un langage dédié à la programmation de systèmes combinant des parties algorithmiques et réactives. Il s'agit d'une extension de ML avec des constructions pour la concurrence inspirées des langages synchrones. Celles-ci permettent d'obtenir une très grande expressivité, mais leur implantation efficace représente un défi.

Dans cet article, nous présentons l'implantation de ReactiveML, de la compilation à l'implantation du moteur d'exécution en OCaml. Nous décrivons également une implantation parallèle en mémoire partagée du moteur d'exécution utilisant le vol de tâches. L'approche choisie permet d'obtenir une exécution efficace même en présence de structures de contrôle complexes. Elle s'étend simplement au cas parallèle avec des résultats expérimentaux prometteurs.

1. Introduction

Lors du dernier concours ICFP, [1] la tâche consistait à deviner des fonctions en posant des questions à un serveur de jeux. On pouvait envoyer les requêtes suivantes au serveur : demander un nouveau problème ; demander la valeur de la fonction à deviner sur 256 entrées ; soumettre une solution. Lors de la soumission d'une solution, si la fonction proposée n'était pas correcte, le serveur fournissait un contre-exemple. Les concurrents avaient cinq minutes pour résoudre chaque problème et il était possible de faire au plus cinq requêtes toutes les vingt secondes.

Une architecture possible pour la programmation d'un joueur est la suivante : (1) un processus consacré à la génération de fonctions qui satisfont la spécification de la fonction à deviner et (2) un processus chargé de la communication avec le serveur de jeux. Le processus de communication demande périodiquement plus d'informations sur la fonction à deviner et dès qu'une solution est proposée par le processus de génération de fonctions, il soumet cette solution au serveur de jeux.

On observe ici la description d'un système qui contient à la fois des parties algorithmiques (génération de fonctions) et réactives (synchronisation des processus et communications avec le serveur). Le langage ReactiveML est dédié à la programmation de ce type de systèmes. Il combine l'expressivité d'un langage à la ML (ici un sous-ensemble d'OCaml) avec des constructions de programmation synchrone pour parler du temps et des événements.

Par rapport aux bibliothèques [2] de threads préemptifs comme le module `Thread` d'OCaml, [3] de threads coopératifs comme Lwt, [4] Async, [5] Muthreads [6] ou de programmation événementielle comme Equeue, [7] ReactiveML propose des constructions de communication et de synchronisation plus

1. `http://icfpc2013.cloudapp.net`
2. Nous citons ici des bibliothèques OCaml, mais la situation est similaire dans la plupart des langages généralistes.
3. `http://caml.inria.fr/pub/docs/manual-ocaml-4.01/libthreads.html`
4. `http://ocsigen.org/lwt`
5. `https://ocaml.janestreet.com/ocaml-core/latest/doc/async_core`
6. `http://christophe.deleuze.free.fr/muthreads`
7. `http://www.camlcity.org/archive/programming/equeue.html`

expressives. Elles permettent en particulier de préempter et de suspendre l'exécution d'un processus à l'émission d'un signal. Ces constructions réactives sont héritées du langage synchrone Esterel [4, 2] qui est dédié à la programmation du contrôle de systèmes embarqués temps-réel critiques.

Dans cet article, nous présentons l'implantation de ReactiveML. Comme habituellement dans les langages fonctionnels, la concurrence est réalisée à l'aide de continuations [20]. L'originalité de l'approche vient de l'utilisation d'une structure de données annexe pour la gestion de l'activation des processus en présence de préemption et de suspension.

Nous commençons par présenter intuitivement la sémantique de ReactiveML à travers des exemples (partie 2). La partie 3 présente la compilation de ReactiveML qui est fondée sur un langage intermédiaire à base de continuations. Le moteur d'exécution est ensuite décrit sans préemption ni suspension (partie 4), puis étendu avec ces constructions (partie 5). La partie 6 présente une implantation parallèle du moteur d'exécution. Enfin, nous concluons par une discussion sur les travaux similaires (partie 7).

2. Présentation de ReactiveML

Reprenons l'exemple du concours ICFP 2013. Les fonctions à deviner prennent un argument qui est un entier 64 bits et le corps de la fonction est composé des constantes entières 0 et 1, de variables, d'un test à zéro, d'opérations unaires, binaires et d'un itérateur. Comme en OCaml, l'arbre de syntaxe abstraite de ce langage peut être représenté à l'aide de types structurés de la façon suivante :

```
type program = { input : ident; expr : expr }
and ident = { name : string; mutable value : Int64.t; }
and expr =
  | Const of Int64.int64
  | Var of ident
  | If_Zero of expr * expr * expr
  ...
```

On peut ensuite définir des fonctions de manipulation de ces valeurs :

```
let rec eval_expr e = match e with
  | Const c -> c
  | Var v -> v.value
  | If_Zero (e1, e2, e3) ->
      if eval_expr e1 = Int64.zero then eval_expr e1 else eval_expr e2
  ...
```

La particularité de ReactiveML vient de ses constructions réactives. On définit ci-dessous un processus `guesser` qui, à partir de la description `pb` d'un problème (un ensemble d'opérateur et une taille de terme) émet sur le signal `guess` des programmes qui respectent la spécification lue sur le signal `spec`. La fonction `random_program` génère aléatoirement des valeurs de type `program` à partir de la définition du problème à résoudre et la fonction `check` vérifie si un programme respecte une spécification.

```
let process guesser pb spec guess =
  loop
    let f = random_program pb in
    if check f (last ?spec) then emit guess f;
    pause
  end
```

ReactiveML est fondé sur le modèle synchrone où le temps est défini comme une succession d'instants logiques. Le mot clé `process` indique que l'exécution de `guesser` peut durer plusieurs instants et

l'expression `pause` marque l'attente de l'instant suivant. La communication entre les processus se fait par des signaux transportant des valeurs. Ici, le corps du processus est une boucle infinie `loop`/`end` qui génère aléatoirement une fonction `f`, récupère sur le signal `spec` avec l'expression `last ?spec` les couples d'entrées/sorties que cette fonction doit respecter et émet `f` sur le signal `guess` si elle convient.

De façon similaire au processus `guesser`, on peut définir un processus `increase_spec` qui, à chaque instant, demande au serveur de jeux plus d'informations sur la fonction à deviner (le module `Webapi` se charge des communications réseaux).

```
let process increase_spec pb spec =
  for i = 0 to Array.length inputs_array - 1 do
    let inputs = inputs_array.(i) in
    let outputs = Webapi.eval pb.id inputs in
    emit spec (inputs, outputs);
    pause
  done
```

On peut maintenant définir le processus `communicator` qui gère les communications avec le serveur. Il est composé de deux parties qui s'exécutent en parallèle. La première, lignes 2 à 7, est une boucle qui attend sur le signal `guess` une fonction `f` calculée par le processus `guesser` et demande au serveur de jeux si cette fonction est une solution au problème. Si c'est le cas, le signal `finished` est émis. Sinon, le contre-exemple fourni par le serveur de jeux est envoyé sur le signal `spec`. La seconde partie du processus, lignes 9 à 18, demande périodiquement plus d'informations au serveur sur la fonction à deviner en exécutant le processus `increase_spec` (la construction `run` marque l'exécution de processus). Pour limiter le nombres de requêtes envoyées au serveur, le processus `increase_spec` est exécuté dans une construction `do`/`when` qui exécute son corps uniquement aux instants où le signal de contrôle (`tick`) est émis. La boucle ligne 15 gère l'émission du signal `tick` toutes les quatre secondes (le processus `wait d` ne fait rien pendant `d` secondes puis termine). Cette boucle est comprise dans une construction `do`/`until` qui préempte l'exécution lorsque le signal `guess` est émit. Enfin, cette construction `do`/`until` est elle même comprise dans une boucle pour réémettre le signal `tick` périodiquement. Lors de la déclaration du signal `tick` (ligne 10), il faut donner une valeur par défaut et une fonction de combinaison. En effet, la valeur d'un signal est le résultat de l'itération (un *fold*) de la fonction de combinaison sur l'ensemble des valeurs émises sur le signal en partant de la valeur par défaut.

```
1  let process communicator pb guess spec finished =
2    loop
3      await guess (f) in
4      match Webapi.guess pb.id f with
5      | Guess_win -> emit finished ()
6      | Guess_mismatch (input, output, _) -> emit spec ([|input|], [|output|])
7    end
8    ||
9    begin
10     signal tick default () gather (fun x y -> ()) in
11     do run increase_spec pb spec when tick done
12     ||
13     loop
14       do
15         loop emit tick (); run wait 4.0 end
16       until guess -> run wait 4.0 done
17     end
18   end
```

Enfin, le processus `solve` essaye de résoudre un problème donné en paramètre en moins de cinq minutes. Pour cela, il exécute le processus `communicator` et le processus `guesser` en parallèle. Pour gérer la terminaison, ces deux processus sont exécutés dans une construction `do/until` contrôlée par le signal `finished`. Ce signal est soit émis par le processus `communicator`, soit par la troisième branche parallèle du processus `solve` au bout de cinq minutes.

```
let process solve pb =
  signal guess default dummy_prog gather (fun x y -> x) in
  signal spec memory [] gather (fun x y -> x :: y) in
  signal finished default () gather (fun x y -> ()) in
  do
    run communicator pb guess spec finished
    ||
    await spec; run guesser pb spec guess
    ||
    run wait (5. *. 60.); emit finished ()
  until finished done
```

On peut constater que la valeur par défaut du signal `spec` est introduite avec le mot clé `memory` au lieu de `default`. Cela signifie qu'au lieu de partir de la valeur par défaut pour calculer la valeur du signal, il faut partir de la dernière valeur du signal. Ainsi, le signal `spec` ne fait qu'accumuler les informations sur la fonction à deviner.

3. Compilation

La compilation de ReactiveML passe par un langage intermédiaire appelé $\mathcal{L}_k$ fondé sur l'utilisation de continuations. Ce langage a été introduit dans [16]. La principale opération pour la traduction de ReactiveML en $\mathcal{L}_k$ est une transformation CPS (*Continuation Passing Style*) partielle. Pour chaque opération potentiellement bloquante comme l'instruction `pause` ou l'attente d'un signal, on crée une continuation contenant la suite du programme. Le reste du code demeure inchangé. Nous nous intéressons dans un premier temps au langage sans préemption ni suspension. Nous verrons dans la partie 5 comment ajouter ces structures de contrôle.

ReactiveML On considère pour cette traduction un noyau non minimal de ReactiveML (sans préemption ni suspension) défini par :

$$\begin{aligned} e ::= {} & x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e \mid \texttt{process}\ e \mid \texttt{run}\ e \mid \texttt{pause} \\ & \mid \texttt{let}\ x = e\ \texttt{and}\ x = e\ \texttt{in}\ e \mid e; e \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \\ & \mid \texttt{emit}\ e\ e \mid \texttt{await immediate}\ e \mid \texttt{await}\ e(x)\ \texttt{in}\ e \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e \end{aligned}$$

Il s'agit d'un lambda-calcul avec appel par valeur, étendu avec la création (`process`) et le lancement (`run`) de processus, l'attente du prochain instant (`pause`), la définition parallèle (`let/and`), la déclaration de signaux (`signal`), l'émission d'un signal (`emit`), l'attente immédiate d'un signal (`await immediate`), l'attente de la valeur d'un signal (`await`) et le test de présence d'un signal (`present`). L'expression `await immediate` s termine instantanément lorsque le signal s est émis. L'expression `present` s `then` e_1 `else` e_2 exécute e_1 instantanément si le signal s est présent ou e_2 à l'instant suivant s'il est absent. Le délai de réaction à l'absence, hérité du langage ReactiveC [6], permet d'éviter les contradictions sur la présence des signaux. À partir de ce noyau, on peut encoder la plupart des autres constructions du langage. On notera _ les variables qui n'apparaissent pas libres dans le corps du `let` et () l'unique valeur de type `unit` :

$$e_1 \,||\, e_2 \triangleq \texttt{let}\ \_ = e_1\ \texttt{and}\ \_ = e_2\ \texttt{in}\ ()$$

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ x = e_1\ \texttt{and}\ \_ = ()\ \texttt{in}\ e_2$$

$$\texttt{let}\ f\ x_1\ \ldots x_p = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ f = \lambda x_1.\ldots.\lambda x_p.e_1\ \texttt{in}\ e_2$$
$$\texttt{let rec}\ f\ x_1\ \ldots x_p = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ f = (\texttt{rec}\ f = \lambda x_1.\ldots.\lambda x_p.e_1)\ \texttt{in}\ e_2$$
$$\texttt{let process}\ f\ x_1\ \ldots x_p = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ f = \lambda x_1.\ldots.\lambda x_p.\texttt{process}\ e_1\ \texttt{in}\ e_2$$
$$\texttt{let rec process}\ f\ x_1\ \ldots x_p = e_1\ \texttt{in}\ e_2 \triangleq \texttt{let}\ f = (\texttt{rec}\ f = \lambda x_1.\ldots.\lambda x_p.\texttt{process}\ e_1)\ \texttt{in}\ e_2$$

Ce noyau est non minimal car il contient par exemple la construction $e_1; e_2$ qui peut être encodée comme $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$. On peut en effet proposer une implantation directe de cette construction bien plus efficace que son encodage.

Le langage intermédiaire $\mathcal{L}_k$ Le langage $\mathcal{L}_k$ est défini formellement par :

$$\begin{aligned}
k ::= {} & \texttt{end} \mid \kappa \mid e_i.k \mid \texttt{present}\ e_i\ \texttt{then}\ k\ \texttt{else}\ k \mid \texttt{run}\ e_i.k && \text{(continuations)}\\
& \mid \texttt{signal}\ x\ \texttt{default}\ e_i\ \texttt{gather}\ e_i\ \texttt{in}\ k \mid \texttt{await immediate}\ e_i.k \mid \texttt{await}\ e_i(x)\ \texttt{in}\ k\\
& \mid \texttt{split}\ (\lambda x.(k,k)) \mid \texttt{join}\ x\ i.k \mid \texttt{def}\ x\ \texttt{and}\ x\ \texttt{in}\ k \mid \texttt{bind}\ \kappa = k\ \texttt{in}\ k\\
e_i ::= {} & x \mid c \mid (e_i, e_i) \mid \lambda x.e_i \mid \texttt{rec}\ x = e_i \mid \texttt{process}\ \Lambda\kappa.k && \text{(expressions instantanées)}\\
& \mid \texttt{signal}\ x\ \texttt{default}\ e_i\ \texttt{gather}\ e_i\ \texttt{in}\ e_i \mid \texttt{emit}\ e_i\ e_i
\end{aligned}$$

Le langage $\mathcal{L}_k$ distingue les continuations k et les expressions instantanées e_i qui correspondent aux expressions ML. La continuation `end` marque la fin du programme, alors que κ est une variable qui peut être substituée par une continuation k. L'expression $e_i.k$ évalue l'expression instantanée e_i puis donne sa valeur à la continuation k. Les expressions `split`, `join` et `def` servent à encoder la composition parallèle synchrone. La construction `split` commence l'exécution du parallèle, le `join` synchronise la terminaison des deux branches et la construction `def` récupère les valeurs retournées par les branches avant d'exécuter la continuation. La variable x introduite dans le `split` et utilisée par le `join` est une variable partagée par les deux branches qui permet de les synchroniser. Le paramètre i, qui ne peut valoir que 1 ou 2, différencie la branche gauche ($i = 1$) de la branche droite ($i = 2$) de la composition parallèle. La construction `bind` permet de nommer les continuations. Les expressions instantanées e_i sont similaires à ReactiveML. La principale différence est que la définition d'un processus prend en argument sa continuation κ introduite par le lieur Λ.

Identification des expressions instantanées Avant de pouvoir traduire le code ReactiveML en $\mathcal{L}_k$, le compilateur doit distinguer les expressions instantanées et les expressions réactives. On utilise pour cela un système de types simple qui permet de garantir certaines propriétés de bonne formation des expressions. En particulier, on souhaite interdire l'utilisation d'expressions réactives comme `pause` à l'intérieur des fonctions. On ne peut les utiliser qu'à l'intérieur d'un processus. Ainsi, la transformation CPS ne concerne que les processus, alors que les fonctions sont conservées telles quelles. Cela permet d'améliorer les performances du code généré, puisque le code OCaml n'est pas modifié.

Cette analyse est définie figure 1 par un jugement de la forme $k \vdash e$ où $k \in \{0, 1\}$. Le prédicat $0 \vdash e$ signifie que e est une expression instantanée (on dit aussi combinatoire). $1 \vdash e$ signifie que e est une expression réactive (on dit aussi séquentielle ou à mémoire en reprenant la terminologie des circuits numériques synchrones). Nous ne discuterons pas des choix faits dans la définition de ce prédicat, puisque cela est déjà fait dans [16]. Nous pouvons tout de même rappeler les points les plus importants :

- $k \vdash e$ signifie que $0 \vdash e$ et $1 \vdash e$. L'expression peut donc être utilisée dans n'importe quel contexte, aussi bien dans une expression instantanée que dans une expression réactive. C'est par exemple le cas des variables, des constantes ou encore de l'application.
- Le corps d'une fonction doit être une expression instantanée (Abs), alors que celui d'un processus peut être une expression réactive (ProcAbs).

$$\frac{}{k \vdash x} \qquad \frac{}{k \vdash c} \qquad \frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash (e_1, e_2)} \qquad (\textsc{Abs})\frac{0 \vdash e_1}{k \vdash \lambda x.e_1} \qquad \frac{0 \vdash e_1 \qquad 0 \vdash e_2}{k \vdash e_1\ e_2} \qquad \frac{0 \vdash e_1}{k \vdash \texttt{rec}\ x = e_1}$$

$$(\textsc{ProcAbs})\frac{1 \vdash e_1}{k \vdash \texttt{process}\ e_1} \qquad \frac{0 \vdash e_1}{1 \vdash \texttt{run}\ e_1} \qquad \frac{}{1 \vdash \texttt{pause}} \qquad \frac{k \vdash e_1 \qquad k \vdash e_2 \qquad k \vdash e_3}{k \vdash \texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e_3}$$

$$\frac{k \vdash e_1 \qquad k \vdash e_2}{k \vdash e_1; e_2} \qquad \frac{k \vdash e_1 \qquad k \vdash e_2 \qquad k \vdash e}{k \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e} \qquad \frac{0 \vdash e_s \qquad 0 \vdash e_1}{k \vdash \texttt{emit}\ e_s\ e_1}$$

$$\frac{0 \vdash e}{1 \vdash \texttt{await immediate}\ e} \qquad \frac{0 \vdash e_1 \qquad 1 \vdash e_2}{1 \vdash \texttt{await}\ e_1(x)\ \texttt{in}\ e_2} \qquad \frac{0 \vdash e_s \qquad 1 \vdash e_1 \qquad 1 \vdash e_2}{1 \vdash \texttt{present}\ e_s\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

FIGURE 1 – Bonne formation des expressions

Traduction de ReactiveML vers $\mathcal{L}_k$ La traduction vers $\mathcal{L}_k$ est définie par une fonction $C_k[e]$ paramétrée par une continuation k. Elle prend en argument une expression ReactiveML e et renvoie une continuation dans le langage $\mathcal{L}_k$. Cette fonction est définie sur la figure 2. Elle utilise la fonction $C[e]$ de traduction des expressions instantanées. Celle-ci ne prend pas en paramètre de continuation puisque la transformation CPS ne concerne que les expressions réactives. Commentons ces fonctions de traduction.

- Si l'expression e est instantanée, c'est-à-dire si $0 \vdash e$, alors on ne doit pas lui appliquer la transformation CPS. On la traduit donc en une expression instantanée $C[e]$, que l'on évalue avant de donner sa valeur à la continuation k.
- Il n'y a pas de séquence en $\mathcal{L}_k$, puisque celle-ci est encodée avec les continuations. Dans le cas de $e_1; e_2$, on traduit d'abord e_2 en lui donnant la continuation k. On traduit ensuite e_1 en utilisant la traduction de e_2 comme continuation.
- Pour le `present`, on utilise la construction `bind` pour définir la continuation partagée par les deux branches. Ce partage est nécessaire pour ne pas recopier la continuation dans les deux branches, ce qui pourrait aboutir à une explosion exponentielle de la taille du code généré.
- La traduction de la construction `let/and/in` utilise la construction `split` pour déclencher l'exécution des deux branches. La continuation de chacune des branches est une instruction `join` qui attend la terminaison de l'autre branche. La continuation des `join` est la continuation κ qui est une instruction `def` qui récupère instantanément les valeurs calculées par les deux branches. On utilise la construction `bind` pour partager κ entre les deux branches.

La traduction des expressions instantanées, définie sur la figure 2b, consiste à paramétrer les processus par une continuation κ, puis à traduire le corps du processus en utilisant la fonction $C_\kappa[.]$, c'est-à-dire en utilisant κ comme continuation du corps du processus. Les autres règles appliquent cette transformation de façon structurelle dans toute l'expression.

Traduction en OCaml La traduction de $\mathcal{L}_k$ en OCaml est immédiate. On associe à chaque construction de $\mathcal{L}_k$ une fonction OCaml (un combinateur). On représente une continuation par une valeur de type `'a step = 'a -> unit`, qui est une fonction attendant une valeur de type `'a`. On représente les expressions instantanées par des glaçons de type `unit -> 'a`. Ainsi, la construction $e.k$ est représentée par le combinateur `rml_compute` défini par :

```
let rml_compute e k = (fun _ -> k (e ()))
val rml_compute : (unit -> 'a) -> 'a step -> 'b step
```

Le combinateur prend en argument une expression instantanée `e` et une continuation `k`. Il renvoie une fonction de transition qui évalue l'expression `e` en lui donnant la valeur `()`, puis donne le résultat à la

$$C_k[e] = C[e].k \quad \text{si} \quad 0 \vdash e \qquad C_k[\texttt{run}\ e] = \texttt{run}\ C[e].k \qquad C_k[e_1; e_2] = C_{C_k[e_2]}[e_1]$$

$$C_k[\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2] = \texttt{bind}\ \kappa = k\ \texttt{in}\ \texttt{present}\ C[e]\ \texttt{then}\ C_\kappa[e_1]\ \texttt{else}\ C_\kappa[e_2]$$

$$C_k[\texttt{await immediate}\ e] = \texttt{await immediate}\ C[e].k$$

$$C_k[\texttt{await}\ e_1(x)\ \texttt{in}\ e_2] = \texttt{await}\ C[e_1](x)\ \texttt{in}\ C_k[e_2]$$

$$C_k[\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e] = \texttt{signal}\ x\ \texttt{default}\ C[e_1]\ \texttt{gather}\ C[e_2]\ \texttt{in}\ C_k[e]$$

$$C_k[\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ e] = \texttt{bind}\ \kappa = (\texttt{def}\ x_1\ \texttt{and}\ x_2\ \texttt{in}\ C_k[e])\ \texttt{in}\ \texttt{split}\ (\lambda y.(C_{\texttt{join}\ z\ 1.\kappa}[e_1], C_{\texttt{join}\ z\ 2.\kappa}[e_2]))\ \text{où } z \text{ frais}$$

(a) Traduction des expressions réactives

$$C[x] = x \qquad C[c] = c \qquad C[(e_1, e_2)] = (C[e_1], C[e_2]) \qquad C[e_1\ e_2] = C[e_1]\ C[e_2]$$

$$C[\lambda x.e] = C[\lambda x.C[e]] \qquad C[\texttt{rec}\ x = e] = \texttt{rec}\ x = C[e] \qquad C[\texttt{process}\ e] = \texttt{process}\ \Lambda\kappa.C_\kappa[e]$$

$$C[\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e] = \texttt{signal}\ x\ \texttt{default}\ C[e_1]\ \texttt{gather}\ C[e_2]\ \texttt{in}\ C[e]$$

$$C[\texttt{emit}\ e_1\ e_2] = \texttt{emit}\ C[e_1]\ C[e_2]$$

(b) Traduction des expressions instantanées

FIGURE 2 – Traduction de ReactiveML vers $\mathcal{L}_k$

continuation `k`. Les combinateurs correspondant aux autres expressions ont une forme similaire :

```
val rml_pause : unit step -> 'a step
val rml_present : (unit -> ('a, 'b) event) -> unit step -> unit step -> 'c step
val rml_await_immediate : (unit -> ('a, 'b) event) -> unit step -> 'c step
val rml_await_all : (unit -> ('a, 'b) event) -> ('b -> unit step) -> 'c step
...
```

La suite de l'article décrit l'implantation de ces différents combinateurs.

4. Implantation du moteur d'exécution

Nous décrivons maintenant l'implantation du moteur d'exécution en OCaml. On trouvera une présentation formelle de la sémantique de $\mathcal{L}_k$ qui suit les mêmes principes dans la partie 7.2 de [16].

Principe de l'exécution Le moteur d'exécution de ReactiveML est un ordonnanceur de tâches. Il utilise un ordonnancement coopératif, où chaque processus doit volontairement rendre la main à l'ordonnanceur pour laisser les autres processus s'exécuter. On dispose d'un ensemble $\mathcal{C}$ de continuations à exécuter dans laquelle l'ordonnanceur vient piocher (on représente cet ensemble par une liste). Certains combinateurs, comme par exemple la composition parallèle, ajoutent des continuations dans cette liste. Il existe également une seconde liste appelée *next* qui contient les processus à exécuter à l'instant suivant. On y ajoute typiquement la continuation de la construction `pause`. L'exécution d'un instant suit l'algorithme suivant :

1. On exécute un instant du programme. Pour cela, on exécute tous les processus dans la liste $\mathcal{C}$ jusqu'à ce qu'elle soit vide.

2. On exécute ensuite la fin de l'instant. On réveille alors les processus qui testent la présence d'un signal absent et ceux qui attendent la valeur d'un signal. On transfère ensuite les processus contenus dans la liste *next*, en attente de l'instant suivant, jusque dans la liste $\mathcal{C}$ des continuations à exécuter. On commence l'exécution de l'instant suivant en revenant à la première étape.

Ces deux phases sont présentes dans la sémantique opérationnelle de ReactiveML [17]. Elle est décrite sous la forme de deux réductions, correspondant respectivement à l'exécution de l'instant et à la fin de l'instant.

Le moteur d'exécution présente deux caractéristiques importantes qui viennent s'ajouter à cet algorithme de base :

- Pour obtenir un interprète efficace, il est capital de faire de *l'attente passive* des signaux. Cela signifie qu'un processus en attente d'un signal ne doit être activé que lorsque le signal est présent : il ne doit pas être réveillé pour vérifier la présence du signal plusieurs fois par instant, ni aux instants où le signal est absent. Pour éviter l'attente active, on associe de façon classique à chaque signal une liste des continuations en attente de ce signal. On réveille ces continuations uniquement lorsque le signal est émis.
- L'autre composante du moteur d'exécution, qui est aussi la plus complexe, est la gestion des préemptions (`do`/`until`) et suspensions (`do`/`when`). L'utilisation d'une liste de continuations à exécuter fait que l'on perd complètement la structure du programme. Cela permet d'obtenir une exécution efficace, mais on a besoin d'une autre structure de données pour gérer l'activation des processus, puisque tous les processus ne sont pas forcément actifs à un instant donné. Nous décrirons cette structure que l'on appelle *arbre de contrôle* dans la partie 5.

Combinateurs L'implantation des différentes fonctionnalités du langage peut se faire à partir d'un nombre limité de primitives d'ordonnancement, définies dans un module appelé `Runtime`. La séparation entre la définition des combinateurs et celle du moteur d'exécution permet de clarifier l'implantation en séparant les différents problèmes (*separation of concerns*). Elle autorise aussi le partage de code entre les différentes versions du moteur d'exécution, notamment entre l'implantation séquentielle et l'implantation parallèle que nous décrirons dans la partie 6.

La figure 3 présente un extrait de l'interface du module `Runtime`. On rappelle que le type d'une continuation qui attend une valeur de type `'a` est `'a step = 'a -> unit`. On définit plusieurs combinateurs correspondant aux différentes opérations dont on a besoin :

- `on_current_instant` exécute une continuation à l'instant courant. Cela revient à l'ajouter dans la liste $\mathcal{C}$.
- `on_next_instant` exécute une continuation à l'instant suivant. Cela correspond typiquement à l'opérateur `pause` et revient à ajouter la continuation dans la liste *next*.
- `on_eoi` exécute une continuation à la fin de l'instant. On l'utilise par exemple pour attendre la fin de l'instant pour connaître la valeur d'un signal.
- `on_event` exécute la continuation immédiatement après l'émission du signal, ce qui correspond à la construction `await immediate`.
- `on_event_or_next` correspond à la construction `present`. Il prend donc en entrée deux continuations. Il exécute la première immédiatement si le signal est émis et la seconde à la fin de l'instant dans le cas contraire.

La définition de certains combinateurs devient immédiate une fois que l'on dispose de ces primitives. Par exemple, la construction `await immediate` *e.k* est traduite par le combinateur `rml_await_immediate_expr`, qui utilise la primitive `Runtime.on_event` (`expr_evt()` déclenche l'évaluation de `expr_evt` qui doit renvoyer un signal). De la même façon, la primitive `on_next_instant` permet d'implémenter simplement l'attente de l'instant suivant avec `pause` :

```
let rml_await_immediate expr_evt k _ = Runtime.on_event (expr_evt()) k ()
let rml_pause k _ = Runtime.on_next_instant k
```

```
module type Runtime = sig
  ...
  (* [on_current_instant f] execute f a l'instant courant *)
  val on_current_instant : unit step -> unit
  (* [on_next_instant f] execute f a l'instant suivant *)
  val on_next_instant : unit step -> unit
  (* [on_eoi f] execute f pendant la fin de l'instant *)
  val on_eoi : unit step -> unit
  (* [on_event ev f] execute f a l'emission de evt *)
  val on_event : ('a, 'b) event  -> unit step ->
  (* [on_event_or_next ev f_ev f_n] execute f_ev si le signal est emis,
     sinon execute f_n a l'instant suivant *)
  val on_event_or_next : ('a, 'b) event -> unit step -> unit step -> unit
  ...
```

FIGURE 3 – Interface du module `Runtime`

Les autres constructions s'obtiennent en composant plusieurs primitives. On peut, par exemple, implémenter un combinateur appelé `on_event_at_eoi` qui exécute une continuation `f` à la fin de l'instant où le signal `evt` est émis :

```
let on_event_at_eoi evt f = Runtime.on_event evt (fun () -> Runtime.on_eoi f)
```

Lorsque le signal `evt` est émis, on ajoute `f` dans la liste des continuations à exécuter à la fin de l'instant. On peut maintenant définir le combinateur `rml_await_all`, qui correspond à la construction `await` $e(x)$ `in` k et qui permet de récupérer la valeur émise sur un signal :

```
let rml_await_all expr_evt k _ =
  let evt = expr_evt () in
  let await_eoi _ =
    let v = Runtime.Event.value evt in
    Runtime.on_next_instant (fun () -> k v)
  in
  on_event_at_eoi evt await_eoi
```

Ce combinateur prend en entrée un signal `expr_evt` et une continuation `k` qui attend la valeur du signal. On utilise le combinateur `on_event_at_eoi`, que l'on vient de définir, pour exécuter la fonction `await_eoi` à la fin de l'instant où le signal `evt` est émis. Cette fonction lit la valeur du signal en utilisant le module `Runtime.Event` qui regroupe les fonctions relatives aux signaux. Plus précisément, on appelle la fonction `value` qui renvoie la valeur du signal à l'instant courant. On demande enfin l'exécution de la continuation `k` à l'instant suivant, en lui donnant comme argument la valeur `v` du signal.

Composition parallèle synchrone Nous allons maintenant décrire l'implantation de la composition parallèle synchrone (`let` $x = e$ `and` $x = e$ `in` e en ReactiveML), qui correspond aux constructions `split`, `join` et `def` en $\mathcal{L}_k$. Le principe est le suivant :

- L'instruction `split` crée un compteur qui est une référence initialisée au nombre de branches. Elle crée aussi une référence par branche pour stocker le résultat de cette branche.
- Les instructions `join` et `def` sont implémentées par le même combinateur. Celui-ci décrémente le compteur, puis stocke le résultat de son argument, qui est la valeur de la branche, dans la référence associée à la branche. Si le compteur atteint zéro, alors toutes les branches ont terminé. Le combinateur appelle alors sa continuation avec un n-uplet formé des valeurs des différentes branches.

Pour obtenir une exécution efficace, le moteur d'exécution définit également un opérateur de composition parallèle n-aire qui reprend les mêmes principes. On peut aller encore plus loin et partager les points de synchronisation de façon dynamique. L'idée est que si la continuation d'une composition parallèle est une instruction `join` d'un autre parallèle, alors on peut utiliser le même compteur pour les deux parallèles. Pour cela, chaque combinateur prend maintenant en argument un point de synchronisation, qui est soit `None`, soit `Some jp` où `jp` est le compteur associé à la composition parallèle dans laquelle est lancé le processus. Lorsque l'on exécute une instruction `split`, on réutilise le compteur donné en argument et on en crée un nouveau si l'argument vaut `None`. Grâce à cette approche, on peut n'utiliser qu'un seul compteur pour synchroniser un nombre dynamique de processus.

Gestion des signaux Pour obtenir une exécution efficace, il est important de ne pas faire d'attente active des signaux, c'est-à-dire que l'attente d'un signal ne doit rien coûter tant que le signal est absent. On associe pour cela deux listes d'attente à chaque signal. Un signal est ainsi un triplet `(n, wa, wp)` où `n` est un enregistrement contenant le statut et les valeurs émises sur le signal, `wa` contient les processus en attente de l'émission du signal et `wp` contient les processus qui testent la présence du signal avec la construction `present`. On réveille les processus dans ces deux listes au moment de l'émission d'un signal. On implémente le combinateur `on_event` par :

```
let on_event (n, wa, wp) f =
  if Event.status n then f () else wa := f :: !wa
```

On teste tout d'abord la présence du signal, c'est-à-dire s'il a déjà été émis au cours de l'instant. Si c'est le cas, alors on exécute la fonction `f`. Sinon, on la stocke dans la liste `wa` des processus en attente de l'émission du signal.

On procède de façon similaire pour la primitive `on_event_or_next`, qui correspond à la construction `present` :

```
let on_event_or_next (n, wa, wp) f_ev f_n =
  let act () = if is_eoi () then next := f_n :: !next else f_ev () in
  if Event.status n then f_ev ()
  else (wp := act :: !wp; weoi := wp :: !weoi)
```

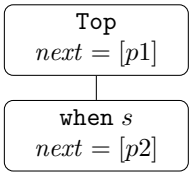
Si le signal est présent, alors on lance la fonction `f_ev` immédiatement. Sinon, on ajoute la fonction `act` dans la liste `wp`. On ajoute ensuite cette liste dans une liste globale appelée `weoi` (qui est donc une liste de références vers des listes). On réveille tous les processus dans ces listes au moment de la fin de l'instant. La fonction `act` teste si l'on est en train d'exécuter la fin de l'instant avec la fonction `is_eoi`. Si ce n'est pas le cas, alors cela signifie que le processus a été réveillé car le signal a été émis. On lance donc immédiatement `f_ev`. Sinon, cela signifie que le signal n'a pas été émis au cours de l'instant et il est donc absent. On ajoute donc `f_n` à la liste *next* des processus à exécuter à l'instant suivant. On peut remarquer qu'il n'est pas nécessaire d'enlever `wp` de la liste `weoi` en cas d'émission car dans ce cas, `wp` contient la liste vide.

5. Suspension et préemption

Nous avons vu dans la partie précédente les bases de l'implantation du moteur d'exécution de ReactiveML. Nous allons maintenant voir comment prendre en compte le reste du langage, et en particulier la suspension et la préemption. Ces deux structures de contrôle différencient ReactiveML des autres bibliothèques de concurrence coopérative mais rendent aussi l'implantation bien plus complexe. Nous nous contentons ici de présenter les grands principes de l'implantation. On trouvera une présentation plus formelle dans la partie 7.3 de [16] qui présente une sémantique de $\mathcal{L}_k$ avec préemption et suspension.

Principe L'utilisation d'une liste $\mathcal{C}$ de continuations à exécuter fait que l'on perd complètement la structure du programme. Cela permet d'obtenir une exécution efficace, mais on a besoin d'une autre structure de données pour gérer l'activation des processus, puisque tous les processus ne sont pas forcément actifs à un instant donné. Cette structure s'appelle *arbre de contrôle*, puisqu'il s'agit d'un arbre n-aire traduisant l'imbrication des préemptions et suspensions dans le programme. Chaque nœud de cet arbre correspond à une construction `do/until` ou `do/when` dans le programme. Considérons par exemple le processus suivant :

```
let process control_tree s p1 p2 =
  emit s; pause; run p1
  ||
  do
    pause; run p2
  when s done
```



Le processus `control_tree` prend en entrée un signal `s` et deux processus `p1` et `p2`. Il lance `p1` et `p2` au second instant, mais `p2` n'est activé qu'aux instants où le signal `s` est présent. La figure sur la droite montre l'arbre de contrôle associé à l'exécution de ce programme à la fin du premier instant de l'exécution de `control_tree`. Sa racine notée `Top` correspond aux processus toujours actifs, alors que le nœud `when` s est associé à la suspension. Chaque nœud de l'arbre de contrôle contient une liste *next* des continuations à exécuter au prochain instant où le corps de la structure de contrôle est actif, c'est-à-dire au prochain instant où le signal `s` est présent dans le cas de la suspension. La liste *next* que l'on a évoquée précédemment est celle qui est associée au nœud `Top` et qui correspond aux processus activés à tous les instants. Ainsi, à la fin du premier instant, la liste *next* de la racine de l'arbre contient le processus `p1`, alors que celle associée à la suspension contient `p2`. On transfère alors les processus depuis la liste *next* de la racine dans la liste $\mathcal{C}$ des processus à exécuter, mais pas ceux du nœud `when` s. On ne le fera qu'à la prochaine émission du signal `s`, lorsqu'on saura que le corps de la suspension est activé.

Dans l'implantation, tous les combinateurs prennent désormais en argument le nœud de contrôle correspondant au contexte dans lequel s'exécute le code. La fonction de traduction $C_k[e]$ (figure 2) est modifiée pour ajouter ces arguments supplémentaires [16]. De même, les primitives `on_next_instant`, `on_event` et `on_event_or_next` de la figure 3 prennent en argument un nœud de l'arbre de contrôle. Par exemple, `on_next_instant ctrl f` exécute `f` à l'instant suivant où le nœud de contrôle `ctrl` est activé, en l'ajoutant dans la liste *next* du nœud `ctrl`.

Implémentation de `do/until` En ReactiveML, la préemption est faible. Cela signifie que l'on ne préempte le corps de la suspension qu'à la fin de l'instant si le signal de préemption est présent. On vérifie donc à la fin de chaque instant si ce signal est présent. Si c'est le cas, alors on désactive le nœud correspondant de l'arbre de contrôle et on ajoute la continuation de la préemption dans la liste *next* du nœud parent. Une alternative est d'utiliser la primitive `on_event` pour attendre l'émission du signal de préemption. On enregistre alors un processus pour désactiver le nœud à la fin de l'instant où le signal est présent.

Implémentation de `do/when` Dans le cas du `do/when`, le nœud de contrôle est activé au moment de l'émission du signal de suspension. On transfère alors les processus depuis sa liste *next* dans la liste $\mathcal{C}$ des processus à exécuter. On active également les nœuds enfants dans l'arbre de contrôle. Pendant la fin de l'instant, on ne doit parcourir un nœud associé à une suspension et ses enfants que si le signal de suspension a été émis pendant l'instant. On désactive alors le nœud pour l'instant suivant et on se remet en attente du signal de suspension.

Attente passive avec preemption et suspension Nous avons vu dans la partie 4 que lorsqu'un processus attend un signal, on le place dans la file d'attente attachée au signal et on le réveille uniquement lorsque le signal est émis. Mais si le processus s'exécute sous une suspension, il ne faut le lancer que si le signal de suspension est présent à cet instant. Dans le cas contraire, tout se passe comme si le processus n'avait pas vu l'émission et il doit rester en attente de la prochaine émission du signal. Illustrons ce problème avec le processus suivant :

```
1  let process await_when =
2    signal act, s in
3    do
4      await immediate s; print_endline "Recu!"
5    when act done
6    || loop pause; emit act; pause end
7    || emit s; pause; emit s
```

On attend l'émission de `s` pour afficher immédiatement le message `"Recu!"`, mais uniquement aux instants où le signal `act` est présent, c'est-à-dire aux instants pairs (voir ligne 6). La première émission de `s`, dans la troisième branche du parallèle, a lieu au cours du premier instant, au cours duquel le corps de la suspension n'est pas actif. On ne doit donc pas réveiller le processus en attente de `s`. On émet ensuite une seconde fois `act` au cours du deuxième instant. Comme le signal de contrôle est cette fois présent, on affiche le message `"Recu!"`.

Une première solution à ce problème, qui est utilisée dans les premières versions de ReactiveML, consiste à utiliser l'attente active lorsque l'on attend un signal sous une suspension ou une préemption active. Une autre approche permet de gérer l'attente passive dans tous les cas. On suit l'algorithme suivant pour lancer une fonction `f` lorsque le signal `evt` est émis et que le nœud de contrôle `ctrl` est actif :

1. Si le signal est présent à l'instant où on démarre l'attente, alors on peut lancer directement `f`. En effet, on sait que le nœud `ctrl` est actif puisque l'on est en train d'exécuter un processus dans ce contexte.
2. Sinon, on met une fonction dans la liste d'attente du signal. Celle-ci effectue les étapes suivantes à l'émission du signal :
   - Si le nœud de contrôle est actif, alors on lance `f`.
   - Sinon, on attend à la fois la fin de l'instant et l'activation du nœud de contrôle `ctrl` qui peut avoir lieu plus tard dans l'instant. En effet, dans le cas d'une expression `do e when s done`, on active le nœud de contrôle correspondant à la suspension lorsque `s` est émis, ce qui peut se passer plus tard dans l'instant. Si le nœud `ctrl` est activé avant la fin de l'instant, alors on lance `f`. Sinon, on sait que le nœud n'est pas actif, ce qui signifie qu'il faut attendre la prochaine émission du signal `evt`. On recommence donc au début de l'étape 2.

Les deux approches ont leurs avantages. Par exemple, l'attente active est plus efficace si le signal `s` est présent à tous les instants mais que `act` est rarement présent. À l'opposé, la version avec attente passive est plus efficace si le signal de suspension `act` est toujours présent et si `s` est rarement émis.

6. Exécution parallèle

Nous cherchons maintenant à répondre à la question de la parallélisation du moteur d'exécution de ReactiveML. La version que nous avons décrite précédemment est une implantation séquentielle de la concurrence avec un ordonnancement coopératif. On souhaite maintenant exécuter un programme ReactiveML sur plusieurs cœurs pour gagner en efficacité, mais sans changer le programme et de façon transparente pour l'utilisateur.

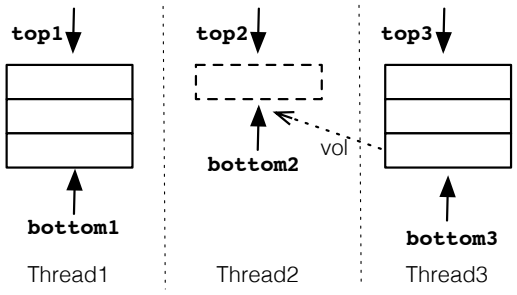


FIGURE 4 – Vol de tâches

Outils et langage ReactiveML est bâti sur un noyau d'OCaml (sans objets, variants polymorphes ni modules). Malheureusement, nous ne pouvons pas utiliser la version standard d'OCaml pour implémenter le moteur d'exécution parallèle. En effet, bien qu'il soit possible de créer des threads en OCaml (avec le module `Thread` de la librairie standard), un verrou global empêche l'exécution de plusieurs threads OCaml en parallèle [8]. Nous avons donc choisi d'utiliser le langage F# [9] pour mener à bien cette expérience. Il est en effet très proche d'OCaml et quasiment compatible au niveau du source avec le sous-ensemble d'OCaml que génère le compilateur ReactiveML.

Implémentation On a vu que le moteur d'exécution de ReactiveML est centré autour d'une liste $\mathcal{C}$ de continuations dans laquelle l'ordonnanceur vient piocher des tâches à exécuter. Le principe de la version parallèle en mémoire partagée est d'avoir plusieurs threads qui viennent piocher des continuations dans la liste $\mathcal{C}$ et les exécutent. On ne parallélise que l'exécution au cours d'un instant. En particulier, la fin de l'instant reste purement séquentielle et tous les threads se synchronisent à la fin de chaque instant.

Pour représenter la liste $\mathcal{C}$, on utilise une structure de données concurrente pour réaliser du *vol de tâches* ou *work stealing* [14]. La figure 4 représente le principe de l'algorithme. Chaque thread dispose de sa propre liste de continuations à exécuter, qui est une file à double entrée ou *deque*, symbolisée par un sommet noté `top` et un pointeur `bottom` indiquant le bas de la file. Lorsque la liste d'un thread est vide, comme dans le cas du second thread de la figure, il va « voler » des tâches à exécuter dans la file d'un autre thread, ici le troisième thread. L'utilisation d'une file à double entrée permet d'éviter dans la majorité des cas les conflits entre le voleur, qui accède au bas `bottom` de la file, et le propriétaire de la file qui accède au sommet `top` de la file. Cet algorithme peut s'implémenter de façon très efficace sans verrous avec des opérations atomiques comme le *compare-and-swap* [10]. Nous utilisons la classe `System.Collections.Concurrent.ConcurrentBag` de la librairie standard .NET qui implémente cette structure de données.

Puisque les threads communiquent par mémoire partagée, on peut réutiliser l'essentiel du code de la version séquentielle du moteur d'exécution. Il faut toutefois gérer les accès concurrents aux différentes structures de données du moteur d'exécution. Le principe est d'associer un verrou à chaque structure partagée. On peut cependant faire mieux :

- On associe à chaque composition parallèle synchrone un compteur du nombre de branches qui ont terminé leur exécution. Comme il s'agit d'entiers, on utilise des opérations atomiques pour décrémenter ces compteurs, de la classe `System.Threading.Interlocked`.
- On peut utiliser des structures de données sans verrous [14] pour les listes partagées. On utilise par exemple une file sans verrous de la classe `ConcurrentStack` pour les listes *next* des nœuds de l'arbre de contrôle.

8. Voir par exemple la partie 19.10.2 *Parallel execution of long-running C code* de `http://caml.inria.fr/pub/docs/manual-ocaml/manual033.html`

9. `http://research.microsoft.com/en-us/projects/fsharp/`

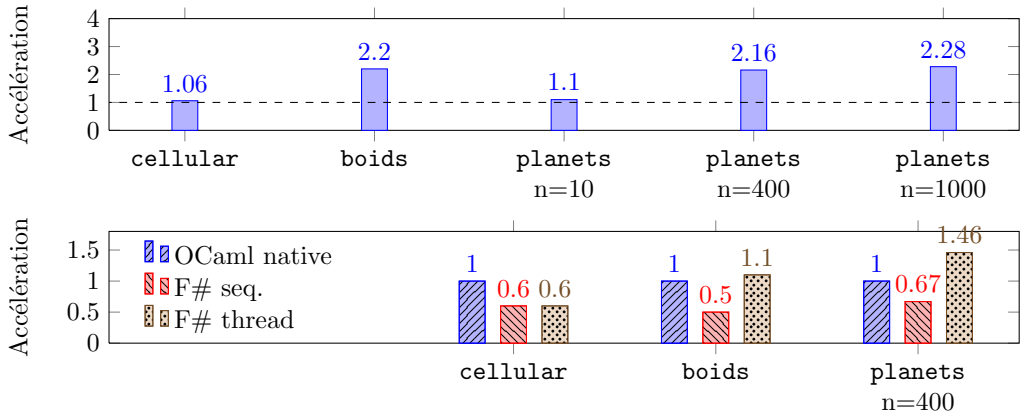


FIGURE 5 – Résultats expérimentaux du moteur d'exécution parallèle en F#
(4 threads, 2 processeurs avec 2 cœurs chacun, F# 3.0)

– On peut se passer de verrous s'il n'y a pas de risques de course critique (*data race*). On parle de course critique si une lecture et une écriture ou deux écritures sur une même location mémoire ne sont pas ordonnées par des verrous. Dans le cas de la dernière valeur d'un signal, elle n'est modifiée que pendant la fin de l'instant, qui est séquentielle, et on ne fait que lire la valeur pendant l'instant. Il n'y a donc pas de risque de course critique.

On utilise des verrous pour les nœuds de l'arbre de contrôle, mais ceci ne pose pas de problème car les modifications de ces nœuds sont rares. L'autre cas où l'on utilise encore un verrou est pour l'implantation des signaux. En effet, au moment de l'émission d'un signal, on doit, de façon atomique, changer le statut du signal et récupérer les processus en attente du signal pour les réveiller. Il faut utiliser un verrou pour garantir qu'aucun autre processus ne puisse voir le signal absent sans être dans la liste des processus en attente.

Résultats expérimentaux La figure 5 montre les performances du moteur d'exécution parallèle écrit en F#. On mesure le temps d'exécution total de plusieurs instants de chaque programme. Le premier graphe montre les résultats pour plusieurs exemples typiques de ReactiveML, tirés de la distribution du compilateur. On mesure ici l'accélération par rapport à la version séquentielle F#, c'est-à-dire le rapport entre la durée du calcul pour la version séquentielle et la durée du calcul dans la version parallèle. Plus la barre est élevée, plus la version parallèle est rapide. Les deux premiers exemples sont des simulations d'automates cellulaires [7] et de *boids* [19], où l'on simule une nuée d'oiseaux en vol. Les trois dernières barres correspondent à une simulation du problème des n-corps dans laquelle on fait varier le nombre de planètes afin de tester plusieurs rapports calcul/synchronisation. On voit que l'on obtient de bons résultats avec une version parallèle deux fois plus rapides environ, sauf dans le cas de la simulation d'automates cellulaires. Cela peut s'expliquer par le fait que ce programme fait très peu de calculs pour chaque processus et aussi par le fait qu'il alloue énormément de mémoire (on alloue une continuation à chaque appel de `pause`), ce qui pose problème puisque le garbage collector de .NET bloque tous les threads.

Le second graphe mesure l'efficacité du moteur d'exécution par rapport à la version OCaml native, qui est cette fois la référence. On peut voir que la version F# séquentielle est toujours plus lente, souvent par un facteur assez grand. Par exemple, elle est deux fois plus lente dans le cas des automates cellulaires ou des boids. L'utilisation du parallélisme permet le plus souvent de combler ce retard, mais n'offre pas des gains importants par rapport à la version OCaml native de référence. Cela relativise donc l'utilité pratique de cette version du moteur d'exécution, en dehors des expérimentations sur la parallélisation du langage.

7. Travaux similaires

Les travaux les plus proches sont ceux autour de Junior [13] et des SugarCubes [9] qui sont des bibliothèques fondées sur le même modèle de concurrence [6]. L'implantation de ReactiveML a d'ailleurs été en particulier inspirée par l'implantation Simple de Junior par Laurent Hazard [12]. Une des différences entre ReactiveML et les autres implantations du modèle réactif est que ReactiveML déstructure le parallélisme et introduit une notion d'arbre de contrôle. Les autres implantations sont des plongements profonds où une représentation explicite de l'arbre de syntaxe abstraite des termes est interprétée. Par ailleurs, la dernière version des SugarCubes [21] supporte l'exécution de programmes OpenCL [10], ce qui n'est pas le cas de ReactiveML.

Par rapport aux Fair Threads [8], qui sont aussi fondés sur le modèle réactif, ReactiveML ne propose pas de support pour la programmation de systèmes globalement asynchrones localement synchrones (GALS), mais dans ce langage, la partie synchrone est moins expressive. En particulier, il n'y a pas de structure hiérarchique dans le parallélisme.

C. Deleuze décrit dans [11] l'implantation d'une bibliothèque de concurrence coopérative reprenant les constructions de ReactiveML. La principale différence avec notre approche concerne la gestion de la préemption et de la suspension. On associe à chaque processus une pile décrivant son contexte d'évaluation, qui correspond aux parents dans l'arbre de contrôle. Cette approche est moins efficace puisqu'il faut reparcourir la pile associée à chaque processus à chaque fin d'instant. Les résultats expérimentaux présentés dans cet article montre que l'ajout de la préemption et la suspension a un impact assez important sur les performances. C'est pourquoi nous n'avons pas cherché ici à comparer l'efficacité de ReactiveML avec celle d'autres bibliothèques de concurrence coopérative en OCaml comme Lwt [22] ou Async, qui ne proposent pas la même expressivité.

Alors que ReactiveML utilise un moteur d'exécution pour obtenir un ordonnancement dynamique, la concurrence d'Esterel est compilée vers du code séquentiel avec ordonnancement statique, après une analyse de causalité préalable pour rejeter les programmes incorrects. Les premières versions du langage compilent les programmes Esterel vers des automates [4], dont la taille est exponentielle en la taille du programme source. La traduction d'Esterel en circuits [1] utilisée dans les versions suivantes permet d'éviter cette explosion combinatoire et de générer aussi bien un programme qu'un circuit à partir du même programme Esterel. D'autres approches [18] se spécialisent dans la génération de code séquentiel et offrent de meilleures performances. Plus récemment, le langage HipHop [5] intègre Esterel dans Hop, qui est un langage *multi-tier* pour la programmation de services Web. Son moteur d'exécution traduit les constructions Esterel en un arbre de syntaxe abstraite (AST) qui est ensuite interprété dynamiquement suivant la sémantique constructive de [3].

8. Conclusion

Nous avons présenté l'implantation du langage ReactiveML. Elle est basée sur l'utilisation de continuations et d'un arbre de contrôle pour gérer l'activation des processus. Son efficacité repose à la fois sur sa transformation CPS partielle qui laisse le code instantané inchangé, le plongement superficiel dans OCaml qui permet un accès en temps constant aux informations du runtime (signaux, points de synchronisation, nœuds de l'arbre de contrôle) et à la déstructuration du parallélisme. Cette approche s'étend à une exécution parallèle basée sur le vol de tâches avec de bonnes performances.

Un axe de recherche pour améliorer les performances du langage est de compiler les parties statiques des programmes en s'inspirant de ce qui est fait en Esterel. Une première approche remplaçant l'ordonnancement dynamique par un ordonnancement statique choisi à la compilation a été expérimentée [15].

10. `http://www.khronos.org/opencl/`

Remerciements Nous souhaitons remercier les rapporteurs des JFLA pour leur relecture attentive, Frédéric Boussinot pour ses conseils et Johannes Kanig avec qui cela a été un plaisir de participer au concours ICFP.

Références

[1] G. Berry. Esterel on hardware, mechanized reasoning and hardware design, 1992.

[2] G. Berry. Preemption in concurrent systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 72–93, 1993.

[3] G. Berry. The constructive semantics of pure Esterel, 1996.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[5] G. Berry, C. Nicolas, and M. Serrano. Hiphop: a synchronous reactive extension for Hop. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, pages 49–56. ACM, 2011.

[6] F. Boussinot. Reactive C: an extension of C to program reactive systems. *Software: Practice and Experience*, 21(4):401–428, 1991.

[7] F. Boussinot. Reactive Programming of Cellular Automata. RR 5183, INRIA, May 2004. `http://hal.inria.fr/inria-00071405`.

[8] F. Boussinot. Fairthreads: Mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, April 2006.

[9] F. Boussinot and J.-F. Susini. The SugarCubes tool box: A reactive Java framework. *Software Practice and Experience*, 28(4):1531–1550, 1998.

[10] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

[11] C. Deleuze. Programmation réactive en OCaml. *Journal Européen des Systèmes Automatisés*, 43(7-8-9):757–771, 2009. 15 pages.

[12] L. Hazard. Simple. An efficient implementation of Junior.

[13] L. Hazard, J.-F. Susini, and F. Boussinot. The Junior reactive kernel. RR 3732, INRIA, 1999. `http://hal.inria.fr/inria-00072933`.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[15] L. Jachiet. Compilation de ReactiveML. Master's thesis, École normale supérieure, 2013.

[16] L. Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6 - Pierre et Marie Curie, 2006.

[17] L. Mandel and M. Pouzet. ReactiveML, un langage fonctionnel pour la programmation réactive. *Technique et science informatiques*, 27(9-10):1097–1128, 2008.

[18] D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling Esterel*, volume 86. Springer, 2007.

[19] C.W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

[20] J.C. Reynolds. The Discoveries of Continuations. *Lisp and Symbolic Computation*, 6(3/4):233–248, 1993.

[21] J.-F. Susini. Les SugarCubes v5. Research report, CNAM, 2013.

[22] J. Vouillon. Lwt: a cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12. ACM, 2008.

Unification des couleurs dans un λ-calcul polychrome

Bernard Paul Serpette [1] & Pascal Manoury[2] & Emmanuel Chailloux[3]

1: Projet INDES,
Inria Sophia-Antipolis Méditerranée,
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
Bernard.Serpette@inria.fr

2: Laboratoire Preuves, Programmes et Systèmes (PPS - UMR 7126)
Université Paris Diderot (Paris 7)
Sorbonnes Paris Cité
Bâtiment Sophie-Germain
F-75205 PARIS Cedex 13
Pascal.Manoury@pps.univ-paris-diderot.fr

3: Laboratoire d'informatique de Paris 6 (LIP6 - UMR 7606),
Université Pierre et Marie Curie (Paris 6)
Sorbonnes Université
4, Place Jussieu, 75005 Paris, France
Emmanuel.Chailloux@lip6.fr

Résumé

Dans cet article nous étendons le λ-calcul bi-chrome présenté aux JFLA 2012 pour y introduire la polychromie[1]. On définit une nouvelle transformation, par β-expansion, qui regroupe les expressions de même couleur, chaque couleur pouvant représenter une unité de calcul. On ne se contente plus de pouvoir expliciter la localité d'un calcul dans un modèle à deux couleurs comme pour les clients-serveurs mais nous pouvons traiter les applications multi-tiers. Les propriétés de correction, de terminaison et de confluence de cette nouvelle transformation sont démontrées à l'aide de Coq.

1. Introduction

Nous avons présenté dans l'article " Séparation des couleurs dans un λ-calcul bichrome "[3] un λ-calcul à deux couleurs permettant d'expliciter une partie de l'évaluation d'un terme en précisant (via la couleur) la localité du calcul. Nous avons pu définir une transformation par β-expansion qui regroupe les expressions de même couleur. Un domaine d'application immédiat a été les clients-serveurs d'application Web, tout particulièrement les environnements de développement Hop [2] et OCsigen [1] qui permettent d'écrire dans un même programme la partie serveur et la partie client. L'une des couleurs représente le code serveur et l'autre le code client (code JavaScript exécuté par le navigateur). La transformation de programme que nous avions proposée permet de regrouper les ilôts de même couleur. La racine de l'arbre d'exécution étant dévolue aux serveurs, la transformation de programme permet d'extraire un code monolithique pour le serveur et, pour le client, une série d'expressions

[1]Ce travail a bénéficié du soutien de l'ANR : projet **PWD** (Programmation du Web Diffus) ANR-09-EMER-009-01.

indépendantes entre-elles. Les propriétés de correction, de terminaison et de confluence de cette transformation ont été démontrées à l'aide de l'assistant de preuves Coq[2]. Cette transformation est indépendante de la sémantique de communication et de synchronisation de l'application.

En fait, les préliminaires de ce travail sur le λ-calcul coloré considéraient un nombre quelconque de couleurs, c'est en observant que la preuve de confluence ne passait pas que nous nous sommes restreints à deux couleurs. La conclusion de [3] montre un exemple avec trois couleurs où la confluence n'était pas établie. La restriction à deux couleurs n'empêche pas de rester fortement lié à Hop et Ocsigen avec le couple client/serveur et donc avec deux sites différents de calcul. Mais on peut aussi attribuer les couleurs à des langages spécifiques, la partie commune de ces langages contenant le minimum pour abstraire les communications. Ici nous choisissons le λ-calcul comme dénominateur commun. Cette vision, où l'on attribue un langage à une couleur, est aussi valide pour Hop et Ocsigen car la partie cliente doit être convertie en JavaScript. Par extension, d'autres langages peuvent intervenir, le cas le plus étudié est la présence d'un langage dédié à l'accès aux bases de données [4], mais on peut aussi envisager des langages utilisant les capacités des accélérateurs graphiques (GPGPU à la OpenCL), des langages dédiés à la musique (comme OpenMusic de l'Ircam)...

Pour permettre la coexistence de plusieurs langages métiers, nous proposons une nouvelle transformation permettant le regroupement polychrome d'expressions. Les définitions sont très proches de la version bichrome aux couleurs près. Ces définitions sont toujours formalisées en Coq[3]. La nouvelle transformation repose toujours sur une β-expansion guidée par le rapprochement chromatique qui nécessite maintenant de tenir compte d'un nombre quelconque de couleurs. Les preuves des propriétés de correction et confluence deviennent plus complexes et sont donc complètement à revoir.

L'étape élémentaire de la transformation va regrouper ensemble deux expressions de même couleur. A ce niveau la propriété la plus importante est la correction : le programme d'origine et le programme transformé doivent avoir le même comportement. Un compilateur répétera cette étape élémentaire jusqu'à trouver une forme normale. Ici, la propriété la plus importante est la confluence : le compilateur va choisir un parcours déterministe de l'arbre pour faire les transformations, la confluence permet d'assurer que le choix du parcours n'a pas d'influence sur le résultat final. Il faut aussi prouver que le compilateur termine. Ce sont les trois théorèmes que nous prouverons dans cet article. Nous mentionnerons en conclusion comment obtenir d'autres propriétés plus générales. La première de ces propriétés est l'optimimalité et correspond au nombre de couleurs dans le résultat final. Cette propriété est obtenue par un artefact sur la couleur de la racine de l'expression principale et dont la preuve semble sans difficulté. La seconde propriété concerne la conservation du critère de terminaison qui s'obtient en considérant une β-expansion compatible avec une stratégie de réduction par valeur. Là encore, la preuve semble sans difficulté.

Cet article est découpé en trois parties. La section 2 reprend les notations du λ-calcul bichrome pour les étendre au calcul polychrome afin de définir la transformation de regroupement des couleurs. La section 3 montre les propriétés de correction, de confluence et de terminaison de cette nouvelle transformation. La conclusion discute de la qualité du regroupement des couleurs et revient sur la nécessité d'employer une β-réduction non-déterministe.

2. Définitions

Nous allons reprendre dans cette section les définitions introduites dans la version bicolore de la transformation [3]. La seule différence notoire est le domaine des couleurs qui passe d'un ensemble à deux éléments aux entiers naturels.

[2]Les anciennes sources sont disponibles sur ftp://ftp-sop.inria.fr/indes/rp/jfla2012.v
[3]Les nouvelles sources sont disponibles sur ftp://ftp-sop.inria.fr/indes/rp/jfla2014.v

2.1. λ-calcul polychrome

Le λ-calcul est classiquement défini avec des variables appartenant à un certain domaine *var*, avec des fonctions que nous noterons $\lambda x.B$ et une application que nous noterons $(@\ F\ A)$. Il se formalise en Coq comme suit :

Définition 1 (`expr`)

```
Variable var:Set.

Inductive expr : Set :=
  Var: var → expr
| Fun: var → expr → expr
| App: expr → expr → expr.
```

Nous utiliserons les notations $\lambda xy.B$ pour $\lambda x.\lambda y.B$, $(@\ F\ A\ B)$ pour $(@\ (@\ F\ A)\ B)$ et **let** $x{=}A$ **in** B pour $(@\ \lambda x.B\ A)$. Les **let**s révèlent l'existence d'un *redex* (*Red*ucible *Ex*pression) qui est la structure sur laquelle s'appuie la β-réduction, le pas de calcul essentiel du λ-calcul.

Nous voulons permettre à l'utilisateur de spécifier, pour chaque expression, le processeur en charge de son évaluation. Pour rester abstrait, les unités de calcul seront représentées par des couleurs. Toutes les expressions possèdent donc une annotation de couleur :

Définition 2 (`color` et `cexpr`)

```
Definition color:= nat.

Inductive cexpr : Set :=
| CVar: color → var → cexpr
| CFun: color → var → cexpr → cexpr
| CApp: color → cexpr → cexpr → cexpr.
```

Pour la notation, les abstractions colorient[4] le constructeur λ ainsi que la variable liée : $\lambda^r x^r.B$, $\lambda^b x^b.B$. Les applications colorient les parenthèses englobantes ainsi que l'opérateur d'application : $(@^r\ F\ A\ )$, $(@^b\ F\ A)$. Les expressions seront coloriées de la couleur de leurs racines, $E^b{=}\lambda^b x^b.(@^r\ x^b\ x^b\ )$, même si cette expression contient des sous-expressions d'une autre couleur. Pour les **let**s, les mots-clés seront coloriés avec la couleur de l'application du redex et la variable avec la couleur de l'abstraction : $\mathbf{let}^r\ x^b{=}A\ \mathbf{in}^r\ B$ dénote $(@^r\ \lambda^b x^b.B\ A\ )$.

Nous n'essayerons pas de définir finement l'évaluateur du λ-calcul coloré, ce qui nécessiterait de clarifier la notion d'unité de calcul, d'expliciter les transferts de données, de formaliser la synchronisation entre ces unités de calcul, etc. Ce travail a été fait dans le cadre de Hop avec une sémantique dénotationnelle [7] et une sémantique opérationnelle [2]. Nous allons plutôt nous appuyer sur une sémantique du λ-calcul : l'interprétation d'une expression colorée E^c sera l'interprétation d'une expression E où E^c et E sont reliées par une transformation T.

Si $[\![.]\!]$ représente la sémantique des expressions du λ-calcul alors la sémantique des expressions du λ-calcul coloré sera définie par $[\![E^c]\!]_c = [\![T(E^c)]\!]$.

Une transformation naïve consisterait à *effacer* les couleurs. Ainsi $\lambda^r x^r.x^r$ se transforme en $\lambda x.x$. Malheureusement, des problèmes de conflits de nom apparaissent, $\lambda^r x^r.\lambda^b x^b.x^r$ se transformerait en $\lambda x.\lambda x.x$ alors que l'intention serait plutôt $\lambda x.\lambda y.x$, car deux variables de même nom mais de couleur différente ne sont pas identiques. Pour résoudre ce conflit, nous supposerons l'existence d'une fonction Ψ de $color \times var$ dans var, et l'effacement des couleurs se fera par la fonction $\downarrow$ définie comme suit :

[4] Pour que le texte reste lisible avec une impression en noir et blanc, nous mettons également en exposant l'annotation de couleur, r pour rouge et b pour bleu, v pour violet...

Définition 3 (`clean`)

$$
\begin{array}{rl}
\textbf{Fixpoint} & \downarrow (E^c{:}cexpr) : expr := \\
& \textbf{match } E^c \textbf{ with} \\
& \mid \mathit{CVar}\ c\ v \Rightarrow \mathit{Var}\ \Psi(c,v) \\
& \mid \mathit{CFun}\ c\ v\ b \Rightarrow \mathit{Fun}\ \Psi(c,v) \downarrow b \\
& \mid \mathit{CApp}\ c\ f\ a \Rightarrow \mathit{App} \downarrow f \downarrow a \\
& \textbf{end.}
\end{array}
$$

Nous imposerons comme contrainte que Ψ soit injective : $\forall\ c_1,v_1,c_2,v_2,\ \Psi(c_1,v_1)=\Psi(c_2,v_2) \Rightarrow c_1,v_1=c_2,v_2$. Ainsi les conflits de nom disparaissent : $\forall\ c_1,c_2,v,\ c_1 \neq c_2 \Rightarrow \Psi(c_1,v) \neq \Psi(c_2,v)$. On peut imaginer, par exemple, que Ψ effectue la concaténation d'un numéro de couleur, d'un séparateur et du nom de la variable.

2.2. Contextes

La transformation que nous allons formaliser dans la section suivante s'appliquera à une sous-expression A d'une expression principale E. Pour déterminer la position de A dans E, ainsi que son éventuel remplacement, nous utiliserons la notion de *contexte* [6]. Comme pour la notion de *chemin* dans un arbre, les contextes sont exprimés comme une liste d'arcs reliant deux sous-expressions. Les arcs des contextes sont plus riches car ils gardent en mémoire, dans leurs structures, les expressions voisines de celle qui est pointée par l'arc:

Définition 4 (`edge`)

$$
\begin{array}{l}
\textbf{Inductive } edge : \textbf{Set} := \\
\mid\ \mathit{GFun} : color \rightarrow var \rightarrow edge \\
\mid\ \mathit{GLeft} : color \rightarrow cexpr \rightarrow edge \\
\mid\ \mathit{GRight} : color \rightarrow cexpr \rightarrow edge.
\end{array}
$$

Par exemple un arc de type *GLeft* est un arc partant d'un noeud d'application vers la fonction mais se souvenant de l'expression argument. Ainsi, à partir d'un arc et d'une expression, il est possible de reconstruire l'expression d'origine :

Définition 5 (`link`)

$$
\begin{array}{l}
\textbf{Definition } link\ (g{:}edge)\ (e{:}cexpr) : cexpr := \\
\quad \textbf{match } g \textbf{ with} \\
\quad \mid \mathit{GFun}\ c\ v \Rightarrow \mathit{CFun}\ c\ v\ e \\
\quad \mid \mathit{GLeft}\ c\ f \Rightarrow \mathit{CApp}\ c\ f\ e \\
\quad \mid \mathit{GRight}\ c\ a \Rightarrow \mathit{CApp}\ c\ e\ a \\
\quad \textbf{end.}
\end{array}
$$

Finalement, un contexte se définit de la même manière qu'une liste d'arcs. On notera $*$ le contexte vide et, si δ est un arc et Δ un contexte, on note $\delta \bullet \Delta$ le contexte dont le premier arc est δ et la suite Δ, voire, plus simplement $\delta\Delta$. On formalise cette définition comme le type inductif:

Définition 6 (`context`)

$$
\begin{array}{l}
\textbf{Inductive } context : \textbf{Set} := \\
\mid\ \mathit{XHole} : context \\
\mid\ \mathit{Edge} : edge \rightarrow context \rightarrow context.
\end{array}
$$

Placer une expression dans un contexte consiste à remplacer le trou du contexte par l'expression reçue en argument. Cette opération se définit par induction simple sur la liste des arcs:

Définition 7 (`put`)

```
Fixpoint  put (c:context) (e:cexpr) : cexpr :=
          match c with
          | XHole ⇒ e
          | Edge g h ⇒ (link g (put h e)
          end.
```

Si Δ est un contexte et e une expression, (put Δ e) sera simplement noté Δ(e). De la même manière, si δ est un arc (link δ e) sera noté δ(e).

La figure 1 montre graphiquement dans quel contexte se trouve l'application de droite de Ω=(@ $\lambda x.$(@ x x) $\lambda x.$(@ x x)). Ce contexte est Δ=(Edge (GRight c $\lambda x.$(@ x x)) (Edge (GFun c x) (XHole)))

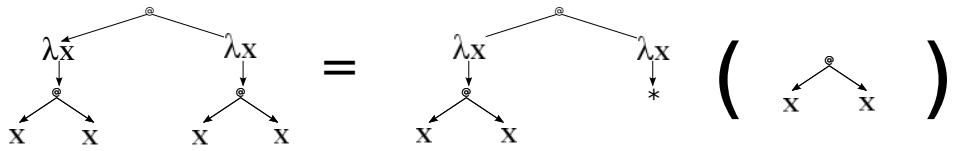


Figure 1: $\Omega = \Delta$ (@ x x)

Un contexte Δ permet donc de repérer une sous-expression A de E, $E = \Delta(A)$, mais permet aussi d'opérer des substitutions. Le remplacement de A par B dans E s'exprime par $\Delta(B)$.

Les arcs seront notés avec leur couleur : δ^r. De la même manière, les contextes non vides sont notés avec la couleur de leurs premier arc : Δ^r.

2.3. Transformation

Le but de la transformation est de réduire la fragmentation des couleurs d'une expression polychrome en conservant la dénotation de l'expression. Le moyen de cette transformation est simplement une β-expansion guidée par le rapprochement chromatique. Par exemple, dans l'expression,

$$(@^b\ (@^r\ F^b\ A\ )\ B^b),$$

les deux expressions F^b et B^b sont séparées par l'application rouge. En sortant F^b de l'application rouge $(@^r\quad)$ par création d'un redex, on obtient

$$(@^b\ (@^b\ \lambda^r x^r.(@^r\ x^r\ A\ )\ F^b)\ B^b),$$

c'est-à-dire, $(@^b\ \lambda^r x^r.(@^r\ x^r\ A\ )\ F^b\ B^b)$, où F^b a rejoint B^b sous une application bleue.

Dans l'exemple ci-dessus, F^b a rejoint B^b en remontant une application. On veut pouvoir également remonter les abstractions. Ainsi, partant d'une expression de la forme

$$(@^b\ \lambda^r y^r.F^b\ B^b)$$

pour que F^b et B^b se rejoignent, on appliquera la transformation de cette expression vers

$$(@^b\ \lambda^r x^r.y^r.x^r\ F^b\ B^b)$$

Pour que ces transformations soient valides, il faut choisir convenablement la variable introduite par le redex: x^r ne doit pas être une variable libre de A, dans le premier cas; x^r doit être différente de y^r et y^r ne doit pas être libre dans F^b, dans le second.

Ces conditions seront réalisées si l'on pose comme condition plus générale que pour remonter une expression bleue F^b au dessus d'une application ou d'une abstraction rouge, il faut que F^b ne contienne *aucune sous-expression rouge*. Quoique plus restrictive, cette contrainte ne bloquera pas le processus de réunion des couleurs. En effet, si F^b contenait une sous-expression rouge, il suffirait de remonter d'abord celle-ci hors de F^b pour pouvoir ensuite faire remonter F^b elle-même hors de son contexte rouge. Par exemple:

$$(@^b\ (@^r\ (@^b\ {F_1}^r\ {F_2}^b)\ A\ )\ B^b)$$

devient

$$(@^b\ (@^r\ \lambda^b x^b.(@^b\ x^b\ {F_2}^b)\ {F_1}^r\ A\ )\ B^b)$$

qui devient à son tour

$$(@^b\ \lambda^r x^r.(@^r\ x^r\ {F_1}^r\ A\ )\ \lambda^b x^b.(@^b\ x^b\ {F_2}^b)\ B^b\ )$$

Si elle n'est pas bloquante, la contrainte de monochromie de l'expression à remonter imposera un certain ordonnancement des applications des transformations, des feuilles vers la racine.

Les chemins de la transformation Donc, pour que la transformation soit possible, il faut avoir une expression E^b d'une couleur donnée ayant une sous-expression A^b, monochrome, de la même couleur. C'est-à-dire que $E^b{=}\Delta(A^b)$. Pour être plus précis, puisque E est colorée en bleu, Δ doit aussi l'être. C'est-à-dire que la couleur du premier arc du chemin qui mène à A^b doit aussi être le bleu: on a donc $E^b{=}\delta^b{\bullet}\Delta'(A^b)$. Ceci suppose que $\Delta{=}\delta^b{\bullet}\Delta'$ *est non vide*. Nous exigerons de surcroît que cette condition soit *maximale*. C'est-à-dire que Δ' ne contienne pas d'arc de couleur bleue, toutefois, Δ' peut être polychrome.

Dans la version bichrome, nous pouvions nous contenter d'une simple alternance de couleurs: dans $\Delta^r(F^b)$, pour remonter l'expression bleue F^b du contexte rouge Δ^r, on créait l'application bleue d'une abstraction rouge: $(@^b\ \lambda^r x^r.\Delta^r(x^r)\ F^b)$. C'est-à-dire, $\mathbf{let}^b\ x^r{=}\mathrm{F}^b\ \mathbf{in}^b\ \Delta^r(x^r)$. Cette solution simple n'est plus valable en présence de chemins polychromes. Si le chemin Δ' est polychrome, il y a de fortes chances que le début et la fin de ce chemin soient de couleurs différentes. Par exemple: $\delta^b{\bullet}\delta^v{\bullet}\delta^r(A^b)$. Pour procéder à la β-expansion de $\delta^v{\bullet}\delta^r(A^b)$, on a le choix entre une variable violette x^v ou une variable rouge x^r. Si l'on choisit la variable rouge x^r, on insère une abstraction rouge entre δ^b et δ^v: $\delta^b(\mathbf{let}^b\ x^r{=}A^b\ \mathbf{in}^b\ \delta^v{\bullet}\delta^r(x^r))$. Si l'on choisit la variable violette x^v, on obtient $\delta^b(\mathbf{let}^b\ x^v{=}A^b\ \mathbf{in}^b\ \delta^v\delta^r(x^v))$ qui remplace l'alternance rouge-bleu par une alternance rouge-violet.

Pour pallier ce problème de prolifération ou de stagnation de la polychromie, nous allons introduire autant de renommages de variables qu'il y a de changements de couleurs, le long de Δ'. Pour notre exemple la transformation sera: $\mathbf{let}^b\ x^v{=}\mathrm{A}^b\ \mathbf{in}^b\ \delta^v(\mathbf{let}^v\ x^r{=}x^v\ \mathbf{in}^v\ \delta^r(x^r))$.

On peut garder le même nom de variable et se contenter d'en changer la couleur. Cette série de renommages met à jour les transferts de contrôle entre les unités de calcul et ainsi, pour notre exemple, que le processeur à la source de δ^v doit servir d'intermédiaire.

Cette série de renommages le long d'un chemin Δ va être générée par la fonction $\Downarrow_{\Delta}^{v^c}$. La couleur c est celle du dernier arc menant à Δ. Implicitement la variable v^c contient la valeur de l'expression remontée. On donne ici la définition de cette fonction.

Définition 8 (`pushdown`)

$$
\begin{array}{l}
\textbf{Fixpoint } \Downarrow_{\Delta}^{v^c} \triangleq \\
\quad \textbf{match } \Delta \textbf{ with} \\
\quad | * \Rightarrow v^c \\
\quad | \delta^g \bullet \Delta \Rightarrow \textbf{let } R{=}\Downarrow_{\Delta}^{v^g} \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{if } c{=}g \\
\qquad\qquad\qquad\qquad \textbf{then } \delta^g(R) \\
\qquad\qquad\qquad\qquad \textbf{else } \{\text{let } v^g{=}v^c \text{ in } \delta^g(R)\} \\
\quad \textbf{end.}
\end{array}
$$

Comme pour la version bichrome, la transformation va s'appliquer sur un chemin reliant deux expressions de même couleur g $\Delta_p(\delta^g(\Delta^c(A^g)))$ et va utiliser $\Downarrow$ après avoir engendré la première liaison pour A. On transformera donc $\Delta_p(\delta^g(\Delta^c(A^g)))$ en $\Delta_p(\delta^g(\{\ let\ v^c{=}A^g\ in\ \Downarrow_{\Delta^c}^{v^c}\ \}))$.

La variable v est choisie de telle sorte qu'elle n'apparaisse pas dans l'expression $\Delta^c(A^g)$, ni libre, ni liée, ni sous quelle couleur que ce soit. Pour satisfaire cette condition, nous supposons l'existence d'une fonction $gensym$: $expr \rightarrow var$ qui sait donner un nom de variable n'ayant pas d'occurrence dans l'expression passée en argument. Il est aisé de définir par induction sur l'expression E^c un prédicat $fresh$ tel que $(fresh\ x\ E^c)$ vérifie que x n'a pas d'occurrence dans E^c.

Ainsi, la variable v créée pour la transformation est générée par $gensym(\Delta^c(A^g))$

La précondition pour que la transformation sur $\Delta_p(\delta^g(\Delta^c(A^g)))$ puisse s'appliquer est que l'expression A^g soit monochrome, c'est-à-dire qu'elle ne contienne pas une sous-expression d'une autre couleur, que l'arc δ^g soit de la même couleur que A^g (implicitement donnée par les annotations de couleur sur δ et A), que le chemin Δ^c ne soit pas vide et que ce chemin ne contienne par la couleur g de A (ce que l'on note $g \notin \Delta^c$). Les prédicats correspondants à ces conditions sont facilement définissables par induction sur les expressions ou les contextes.

En résumé, la transformation est donnée par:

Définition 9 (`hstep`) *$E_1 \nearrow E_2$ si et seulement si:*

`hstep_step` *si $E_1 = \delta^g(\Delta^c(A^g))$ avec Δ^c non vide, si A^g est monochrome (de couleur g) et si $g \notin \Delta^c$ alors $E_1 \nearrow \delta^g(\{\text{let } v^c{=}A^g \text{ in } \Downarrow_{\Delta^c}^{v^c}\ \})$ où $v{=}gensym(\Delta^c(A^g))$.*

`hstep_link` *si $E_1 = \delta(E_1')$, alors, pour toute expression E_2', si $E_1' \nearrow E_2'$, alors , $E_1 \nearrow \delta(E_2')$.*

Cette définition est implémentée par le prédicat inductif `hstep`.

3. Propriétés

La transformation $\nearrow$ a pour ambition d'être intégrée dans un compilateur. Ce dernier va enchaîner les étapes de transformation jusqu'à trouver un état stable (forme normale). D'une part, il faut assurer la correction de la transformation, c'est-à-dire que le programme transformé a le même comportement que le programme d'origine (correction: 3.1). Ensuite, pour soulager l'écriture du compilateur, il faut montrer que l'ordre dans lequel on enchaîne les transformations a peu ou pas de conséquences (confluence: 3.2). Enfin, il faut prouver que l'état stable est toujours atteignable (terminaison: 3.3)).

La polychromie nous a conduit à remanier les preuves de manière importante. Ne subsistent des scripts de preuves du calcul bichrome que 110 lignes qui correspondent aux définitions communes

de bases, sur les 2014 lignes de script. Nous présentons dans cette section les grandes lignes de ces preuves en mettant principalement l'accent sur la gestion des renommages par coloration des variables le long des chemins de transformation (fonction `pushdown`: 8). Pour le résultat de confluence, nous indiquons pourquoi et comment, pour mener à bien les preuves d'existence, il a été plus efficace de définir des fonctions explicites de construction des expressions réclamées par la confluence.

Pour les lemmes et théorèmes énoncés dans cette section, nous indiquons les noms correspondant dans le script Coq pour le lecteur intéressé par les détails des scripts de preuve.

3.1. Correction de la transformation

La transformation $E_1 \nearrow E_2$ procède à la β-expansion de l'expression à transformer, plus précisément à une série de β-expansions (pour toute relation R, nous noterons R^+ sa fermeture transitive et R^* sa fermeture réflexive et transitive). Il y a en effet deux catégories de β-expansion: l'une destinée au regroupement des sous-expressions colorées; l'autre, par effet induit de la multiplicité des couleurs[5], permet, par renommage, d'aller insérer une variable d'une couleur donnée à travers un contexte polychrome en conservant l'alternance des couleurs du contexte (fonction $\Downarrow$).

Théorème 1 (`hstep_sound`) $E_1 \nearrow E_2 \Rightarrow\downarrow E_2 \rightarrow_\beta^+\downarrow E_1$.

Preuve par induction sur la définition de $E_1 \nearrow E_2$.

`hstep_link` il faut montrer que $\downarrow \delta(E_1) \rightarrow_\beta^+\downarrow \delta(E_2)$ si $\downarrow E_1 \rightarrow_\beta^+\downarrow E_2$ pour tout arc δ. Ce que l'on obtient par induction sur les cas de β-réduction pour chaque cas d'arc. C'est établi par le lemme `rt_beta_link`;

`hstep_step` il faut montrer que la transformation effective est bien une β-expansion, c'est-à-dire que $E_1(\{let\ v^c{=}A^g\ in\ \Downarrow_{\Delta^c}^{v^c}\ \})\rightarrow_\beta^+\ \downarrow\ \delta^g(\Delta^c(A^g))$ lorsque δ^g, Δ^c et A^g vérifient les prémisses de la transformation. C'est l'objet du lemme `step_sound`.

Il y a deux manières d'aborder l'induction pour s'occuper de Δ^c. La première, plus intuitive, suivrait l'ordre des évaluations d'un appel par valeur, le pas d'induction serait **let** v^{c_1}=A **in** **let** v^{c_2}= v^{c_1} **in** B $\rightarrow_\beta$ **let** v^{c_2}=A **in** B, mais des difficultés apparaissent dues à la présence potentielle de la variable v^{c_i} dans B. Il est plus facile de défaire les **let** du bas vers le haut avec un pas d'induction **let** v^{c_2}= v^{c_1} **in** B $\rightarrow_\beta$ subst(B,v^{c_2},v^{c_1}), ce qui correspond au renommage de la fonction $\Downarrow$. Ce pas d'induction est donné par le lemme qui suit.

Lemme 1 (`beta_pushdown_var`) $v \notin \Delta \Rightarrow \forall c, \downarrow \Downarrow_\Delta^{v^c} \rightarrow_\beta^* \downarrow \Delta(v_c)$

Tous les problèmes liés à la correction de la transformation viennent de la substitution engendrée par la β-réduction : **let** x=A **in** B $\rightarrow_\beta$ subst(B,x,A). La définition générale de la substitution (par exemple page 146 de [8]) sur les abstractions fait apparaître une nouvelle variable : (subst λ v.B, x,A) = λ z.subst(subst(B,v,z),x,A) où z n'est libre ni dans A, ni dans B. Cette nouvelle variable provoque des casse-têtes dans la preuve. Pour contourner le problème, on introduit des cas particuliers pour la substitution. Par exemple le cas où x=v et le cas où x n'est pas libre dans B dispensent de faire la substitution dans le corps de l'abstraction. Plus généralement la convention de Barendregt [9] où v n'est pas libre dans A permet de se dispenser de l'introduction de la nouvelle variable. La difficulté de la preuve réside maintenant dans le fait de certifier que les substitutions restent dans ces cas particuliers. Lors des renommages, **let** v^{c_2}= v^{c_1} **in** B $\rightarrow_\beta$ subst(B,v^{c_2},v^{c_1}) la convention de Barendregt (x n'est pas libre dans v^{c_1} pour un λx se trouvant dans le chemin Δ) est respectée par le fait que $v \notin \Delta$ dans les prémisses du Lemme 1. Pour le redex principal *let* $v^c{=}A^g$ *in* B, cette

[5] Ce phénomène était absent de la version bicolore.

convention est respectée par le fait que A^g est monochrome et que le chemin Δ ne contient pas la couleur g, donc la couleur de x est différente de celle de A et donc ne peut pas apparaître libre dans celle-ci.

3.2. Confluence

La confluence permet de certifier que si deux transformations E_1 et E_2 sont possibles à partir d'une même expression E, alors faire a priori une des transformations n'empêchera pas de faire l'autre transformation a posteriori. Lorsque l'on n'obtient pas directement des expressions égales, on montre que E_1 et E_2 se transforment en deux expressions R_1 et R_2 pour lesquelles nous avons défini une α-équivalence *ad hoc*, notée $R_1 \equiv R_2$.

Théorème 2 (`hstep_confluence`) $E \nearrow E_1 \ \wedge \ E \nearrow E_2 \Rightarrow$
$E_1 = E_2$
$\vee \ \exists R_1, R_2, \ E_1 \nearrow R_1 \ \wedge \ E_2 \nearrow R_2 \ \wedge R_1 \equiv R_2.$

Dans la version monochrome, nous avions pris l'option d'un lemme permettant de comparer deux contextes Δ_1 et Δ_2 tels que E=$\Delta_1(A_1)$=$\Delta_2(A_2)$, les principaux cas sont donnés dans la figure 2. Cette étude de cas correspond à la méthode que l'on rencontre le plus souvent pour prouver la confluence du λ-calcul. Le cas central, où l'un des redex est inclus dans l'autre, est le cas *compliqué* du λ-calcul, alors que pour notre transformation cela correspond à un cas impossible. C'est pour cette raison que nous avons pris cette approche dans la version monochrome. Pour tous les cas différents de la comparaison, on donnait les expressions R_1 et R_2 et on montrait, cas par cas, la confluence. Sachant que $\Delta 1$ est de la forme $\Delta_p(\delta^g(\Delta^c(*)))$, beaucoup de cas particuliers apparaissent, sans compter ceux qu'engendrent l'introduction de $\Downarrow$.

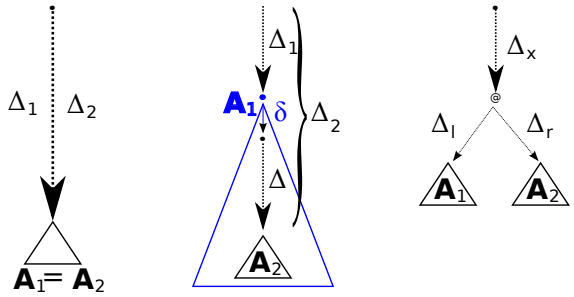


Figure 2: principaux cas de comparaison de deux contextes ayant même racine

Pour la version polychrome, nous avons fait le choix d'exhiber les fonctions qui construisent les expressions R_1 et R_2 réclamées par la confluence. On sait que : $E = \Delta_{p_1}(\delta_1(\Delta_1(A_1)) = \Delta_{p_2}(\delta_2(\Delta_2(A_2))$. Pour la première transformation on a : $E_1 = \Delta_{p_1}(\delta_1(\{\ let\ v{=}A_1\ in\ \Downarrow^v_{\Delta_1}\ \}))$. Il faut donc pouvoir réécrire cette expression E_1 sous une forme : $E_1 = \Delta_{p_{12}}(\delta_{12}(\Delta_{12}(A_2))$ pour que la transformation sur A_2 puisse avoir lieu.

Nous avons défini la fonction $\mathcal{C}$ qui, à partir des informations $\Delta_{p_1}, \delta_1, \Delta_1, A_1, \Delta_{p_2}, \delta_2, \Delta_2$, va produire le triplet : $\Delta_{p_{12}}, \delta_{12}, \Delta_{12}$. A_1 est nécessaire pour calculer le nom de la variable générée pour lier cette expression, i.e. $gensym(\Delta_1(A_1))$. Schématiquement, la fonction $\mathcal{C}$ va parcourir simultanément Δ_{p_1} et Δ_{p_2}, selon les cas du chemin le plus court entre les deux, $\mathcal{C}$ va parcourir ensemble Δ_{p_1} avec Δ_2 ou Δ_{p_2} avec Δ_1, ces trois parcours finissent par une comparaison entre Δ_1 et Δ_2.

Cette fonction $\mathcal{C}$ n'est pas forcément facile à écrire (du moins correcte du premier coup). De fait, nous avons défini trois fonctions principales chargées de construire le préfixe $\Delta_{p_{12}}$ (`build_prefix`), l'arc

δ_{12} (`build_edge`) et le contexte Δ_{12} (`build_path`). Elles font appel à 5 autres fonctions auxiliaires chargées chacune d'une configuration de mise en parallèle des chemins. Toutefois, cette complexité combinatoire reste sans comparaison avec la difficulté à expliciter, dans les lemmes ou via les scripts de preuve, les cas particuliers pour comparer les chemins menant à A_1 et à A_2.

On remarque que $\mathcal{C}$ permet de calculer autant R_1 que R_2, les deux expressions réclamées par la confluence.

Reste maintenant à prouver la correction de $\mathcal{C}$, c'est-à-dire que les valeurs qu'elle calcule permettent bien de retrouver l'expression résultant de la première transformation :

Lemme 2 (Correction de C : `build_premisse_correct`)
$\mathcal{C}(\Delta_{p_1}, \delta_1, \Delta_1, A_1, \Delta_{p_2}, \delta_2, \Delta_2) = (\Delta_{p_{12}}, \delta_{12}, \Delta_{12})$

$\Rightarrow \Delta_{p_1}(\delta_1(\{ \text{ let } v{=}A_1 \text{ in } \Downarrow^{v}_{\Delta_1} \}\,)) = \Delta_{p_{12}}(\delta_{12}(\Delta_{12}(A_2))$

Preuve *Sans problème particulier, il faut suivre la fonction $\mathcal{C}$, par induction lorsqu'elle est récursive, et vérifier l'égalité au fur et à mesure. Ce lemme permet surtout de mettre au point la fonction C.*

Il faut aussi prouver que l'arc et le deuxième contexte produit par la fonction $\mathcal{C}$ satisfont les prémisses exigées par la transformation (lemme `build_premisse_is_premisse`) : δ_{12} a la même couleur que A_2, que Δ_{12} est non vide et ne contient aucun arc de la couleur de A_2. Comme pour le lemme précédent, cela ne pose pas de problème particulier.

L'équivalence *ad hoc* Reste à prouver l'équivalence entre les deux résultats R_1 et R_2. A partir de $\mathcal{C}(\Delta_{p_1}, \delta_1, \Delta_1, A_1, \Delta_{p_2}, \delta_2, \Delta_2)$ on trouve $(\Delta_{p_{12}}, \delta_{12}, \Delta_{12})$ qui permet de construire R_1; à partir de $\mathcal{C}(\Delta_{p_2}, \delta_2, \Delta_2, A_2, \Delta_{p_1}, \delta_1, \Delta_1)$ on trouve $(\Delta_{p_{21}}, \delta_{21}, \Delta_{21})$ qui permet de construire R_2. L'équivalence va être établie en parcourant simultanément les deux triplets. Le cas le plus particulier est lorsque $\Delta_{p_{12}}\delta_{12} = \Delta_{p_{21}}\delta_{21}$. On est alors dans la situation où A_2 est *dans* Δ_{12} et A_1, *dans* Δ_{21}, sans que A_1 et A_2 soient sous-expressions l'une de l'autre; ce peut, par exemple, être les deux termes d'une application. En remontant A_1, puis A_2 on obtient une expression de la forme **let** $v_{11} = A_1$ **in let** $v_{12} = A_2$ **in** B_1; en remontant A_2 puis A_1, on obtient **let** $v_{22} = A_2$ **in let** $v_{21} = A_1$ **in** B_2. Cette situation se retrouve à chaque commutation de couleur par la fonction $\Downarrow$ dans la partie commune de Δ_{12} et Δ_{21}. Pour vérifier que les renommages sont correctement effectués, on pose que ces deux expressions doivent être équivalentes.

De manière générale, l'équivalence que nous posons ressemble à un *renommage explicite* des variables qui nous donnera les cas *ad hoc* d'α-équivalence pour les transformations opérées. En fait, nous avons spécifié trois équivalences

`let_equiv` la première, est utilisée pour R_1 et R_2 et ne va pas prendre en compte de renommage;

`let_equiv_rn1` la deuxième, va être utilisée dès que l'on va rencontrer δ_1 ou δ_2 et va prendre en compte un des deux renommages, *en dur* dans l'équivalence, $E_1 \equiv_{v_{11}=v_{21}} E_2$, pour exprimer que la variable v_{11} de E_1 est renommée v_{21} dans E_2;

`let_equiv_rn2` la dernière équivalence sera utilisée lorsque les deux arcs δ_1 et δ_2 auront été trouvés, cette équivalence va prendre en compte les deux renommages.

Certes, cette technique est particulièrement *ad hoc* mais elle permet d'alléger les cauchemars liés à l'α-équivalence.

3.3. Terminaison

Comme pour la version bichrome, la terminaison est prouvée par la décroissance du nombre d'arcs reliant deux expressions de couleurs différentes. Ce nombre d'arcs est calculé par la fonction B suivante :

Fixpoint B(e) $\stackrel{\triangle}{=}$
  match e **with**
  | CVar c v ⇒ 0
  | CFun c v b^{c_b} ⇒ B(b) + (c ≠ c_b)
  | CApp c f^{c_f} a^{c_a} ⇒ B(f) + (c ≠ c_f) + B(a) + (c ≠ c_a)

Où l'opérateur ≠ est défini par : $c_1 \neq c_2 \stackrel{\triangle}{=}$ **if** c_1==c_2 **then** 0 **else** 1.

Théorème 3 (Fonction décroissante) *$E_1 \nearrow E_2 \Rightarrow B(E_1) = B(E_2) + 1$.*

Pour la version polychrome, il faut montrer que ⇓ n'introduit pas de rupture de couleur. On montre que lors des renommages, **let** $v^{c_2} = v^{c_1}$ **in** B, la couleur c_1 est celle de l'arc entrant et la couleur c_2 est celle de B.

4. Conclusion

Nous avons montré comment une série de β-expansions pouvait regrouper les couleurs dans un λ-calcul polychrome. Nous n'avons pas abordé le problème de l'optimalité, en présence de n couleurs, l'optimal serait d'obtenir n composantes de couleurs différentes. Cet optimal n'est en général pas atteint. En premier lieu, parce que deux sous-expressions de même couleur n'ayant pas un lien commun (par exemple deux expressions rouges sous une racine bleue) ne peuvent pas être regroupées. En second lieu, parce qu'une composante orpheline va empêcher toutes les expressions l'incluant d'être candidates à une transformation (car ne pouvant satisfaire le critère de monochromie). Par exemple un chemin comportant en alternance des arcs de couleurs bleue et rouge et finissant par un arc de couleur violette sera irréductible.

Une première solution pour résoudre ce problème serait d'alléger le critère de monochromie de l'expression à remonter. Le seul critère important est que l'ensemble des couleurs de l'expression à remonter et l'ensemble des couleurs sur le chemin de remontée de l'expression aient une intersection vide, autrement dit, si $E = \Delta_p(\delta^r(\Delta^b(A^r)))$, alors Δ^b, et A^r n'ont pas de couleur en commun.

Une seconde solution, qui semble meilleure, serait de considérer la racine de l'expression comme potentiellement de toutes les couleurs. Cette racine représenterait l'ensemble des expressions *top-level* du programme. Il n'y aurait ainsi plus d'expression orpheline en tant que sous-expression et le cas bloquant évoqué précédemment n'existerait plus. On pourrait voir ainsi la transformation comme une traduction de Hop ou Ocsigen vers Links [5] qui est aussi un langage où l'on précise les lieux d'exécution, mais les annotations ne sont valides qu'au niveau des formes top-level.

D'un autre côté, nous avons vu que le problème principal des preuves vient de la substitution. La substitution n'intervient que par le fait que l'on considère la β-réduction non déterministe où l'on ne précise pas comment on choisit la sous-expression à réduire. La β-réduction non déterministe est indispensable pour la preuve de correction, car la transformation nécessite une réduction forte, c'est-à-dire qu'il faut accepter de réduire sous les abstractions pour retrouver l'expression originale. En d'autres termes, la transformation ne préserve pas la terminaison pour un interprète en appel par valeur, il est possible d'avoir une expression dont le calcul termine mais dont l'expression transformée boucle. En suivant le principe de simulation d'un évaluateur paresseux par un évaluateur par valeur, il est possible de remonter une encapsulation de l'expression par une fonction sans argument, (un *thunk*), ainsi au lieu de remonter A, on remonte λ ().A, et symétriquement, au lieu de remplacer A par une variable, on applique directement cette variable (activation du *thunk*). Dans ce cadre, la preuve de correction se ferait sur un évaluateur standard en appel par valeur, sans référence à la problématique substitution.

Remerciements. Les auteurs remercient Christine Huet et Jérémie Salvucci pour leur relecture attentive ainsi que les rapporteurs de l'article pour leurs remarques constructives.

Bibliographie

[1] Vincent Balat, Jérôme Vouillon, et Boris Yakobowski. Experience Report: Ocsigen, a Web Programming Framework. In Graham Hutton et Andrew P. Tolmach, editeurs, *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming (ICFP)*, pp. 311–316. ACM, 2009.

[2] Gérard Boudol, Zhengqin Luo, Tamara Rezk, et Manuel Serrano. Towards Reasoning for Web Applications: an Operational Semantics for Hop. In *Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*, APLWACA '10, pp. 3–14, New York, NY, USA, 2010. ACM.

[3] Emmanuel Chailloux et Bernard Serpette. Séparation des couleurs dans un lambda-calcul bichrome. In *Journées Francophones des Langages Applicatifs*, January 2012.

[4] James Cheney, Sam Lindley, et Philip Wadler. A practical theory of language-integrated query. In Greg Morrisett et Tarmo Uustalu, editeurs, *ICFP*, pp. 403–416. ACM, 2013.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, et Jeremy Yallop. Links: Web Programming Without Tiers. In *FMCO*, pp. 266–296, 2006.

[6] James H. Morris. *Lambda Calculus Models of Programming Languages.* PhD thesis, Massachusetts Institute of Technology, 1968.

[7] Manuel Serrano et Christian Queinnec. A Multi-tier Semantics for Hop. *Higher-Order and Symbolic Computation*, pp. 1–23, 2010.

[8] Kenneth Slonneger et Barry L. Kurtz. *Formal syntax and semantics of programming languages - a laboratory based approach.* Addison-Wesley, 1995.

[9] Christian Urban, Stefan Berghofer, et Michael Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, CADE-21, pp. 35–50, Berlin, Heidelberg, 2007. Springer-Verlag.

Une sémantique statique pour MongoDB

Adrien Husson

ENS Cachan

Résumé

Les bases de données dites "NoSQL" n'offrent généralement aucune garantie quant au format du contenu de leurs collections ; le soin est laissé à l'administrateur de vérifier la cohérence de ses données et au programmeur d'écrire des requêtes compatibles avec celles-ci.

Ces problèmes ont été résolus dans le contexte d'XML et plus récemment dans le contexte du langage NoSQL Jaql grâce a l'utilisation simultanée du sous-typage sémantique (particulièrement adapté a la définition des types de collections hétérogènes) et d'un calcul de combinateurs récursifs polymorphes appelé filtres.

Nous montrons dans cet article comment utiliser les filtres pour donner une sémantique dynamique et statique aux requêtes de MongoDB, une base de données NoSQL, sans nécessiter aucune annotation de type de la part du programmeur. L'encodage proposé ici offre une formalisation du comportement des requêtes MongoDB et une inférence de type précise sur leur éxécution, ce qui ouvre la voie à de nouvelles garanties statiques pour le programmeur Mongo

1 . Introduction

Le stockage et la manipulation de grandes quantités de données a traditionnellement été fait au moyen d'outils appelés *bases de données SQL*. Une base de donnée contient plusieurs *tables*, des tableaux dont chaque ligne correspond à une entrée, et dont chaque colonne correspond à un *champ* de l'entrée. Les colonnes sont caractérisée par leur nom et par un *type*, auquel tous les éléments de la colonne doivent correspondre ; par exemple "entier", ou "chaîne de caractères". L'ensemble de ces colonnes munies d'un type est appelé *schéma* d'une table et offre un point de vue abstrait sur cette table : chaque entrée, ou ligne, de la table est un n-uplet de valeurs et le type de chacune est celui de la colonne qui lui correspond.

Récemment, un autre type de bases de donnée a émergé pour faciliter le traitement de données hétérogènes difficiles à décrire par les *schémas* décrits plus haut. Moins semblables entre elles que ne le sont les bases de données SQL, elles ont été appelées bases de données *NoSQL*, par opposition à ces dernières. Contrairement aux bases de données SQL, leurs *tables*, ou *collections*, n'embarquent pas de *schéma* : deux entrées d'une collection peuvent contenir des valeurs de types différents. De plus, leur structure est imbriquée, au contraire des tables SQL dont les valeurs ne peuvent pas être elles-mêmes des tables.

Par exemple, les deux éléments suivants pourraient se trouver dans une collection NoSQL : `{tx: "no", success: false}` et `{tx: 2361}`. Chaque élément est un ensemble de paires champ-valeur, mais tandis que le champ `tx` de l'un est une chaîne de caractères, celui de l'autre est un entier ; de plus, le premier élément possède un champ `success` qui est absent du second.

La syntaxe utilisée ci-dessus est celle de JSON, un langage de représentation de données, utilisé par MongoDB [2], un système de base de données NoSQL. MongoDB permet de gérer des collections hétérogènes au moyen d'opérations classiques comme la recherche, la mise à jour, la projection et l'aggrégation, mais n'offre aucun point de vue abstrait, ni sur ses données, ni sur ses requêtes : il n'existe pas de description synthétique du contenu d'une collection MongoDB, et la compatibilité d'une requête avec une collection ne peut être mesurée qu'en éxécutant cette dernière. Par exemple, si une recherche

dans une table SQL porte sur un champ absent de cette table, la base de donnée pourra, par simple inspection du schéma, renvoyer une erreur. MongoDB doit parcourir sa collection et tester la présence du champ sur chacune de ses entrées.[1]

Notre objectif est, étant donné une requête (recherche, mise à jour, ...) d'obtenir une représentation synthétique des champs et types de valeurs que devront avoir les éléments d'une collection MongoDB compatibles avec cette requête. Nous appelons cette représentation synthétique un *type* et demandons à notre solution d'avoir les propriétés suivantes: *a)* un type devra être capable de représenter une liste hétérogène, et *b)*, la méthode par laquelle nous générons une représentation synthétique (un type) devra se satisfaire des requêtes dénuées de toute annotation supplémentaire (nous souhaitons pouvoir appliquer la méthode à des bases de code déjà existantes).

Un langage appelé *filtres* a été développé dans [6] pour, étant donné une transformation sur des documents *XML*, obtenir le type des documents compatibles avec cette transformation : il s'agit d'un calcul de combinateurs récursifs du premier ordre permettant d'encoder des itérateurs sur des listes et des arbres. Le typage ne se fait pas sur les transformations, c'est à dire sur les *filtres* eux-mêmes mais sur l'application d'un filtre à un type d'entrée particulier[2].

Les filtres ont ensuite été adaptés dans [4] afin d'exprimer et de typer la plupart des langages de requêtes NoSQL à travers une étape où une requête est traduite en un filtre, ce qui offre ainsi aux langages traduits *a)* une sémantique formelle et *b)* une inférence de type sur l'éxécution de requêtes écrites dans ces langages. Avec une représentation synthétique des données existantes, il devient possible d'offrir statiquement des garanties concernant l'éxécution des requêtes et le format des données suite à leur éxécution. Par exemple, on pourra garantir l'absence d'un certain champ suite à l'éxécution d'une requête. Nous nous proposons ici de traduire MongoDB vers cette version des filtres développée dans [4]

Pourquoi MongoDB ? D'une part, MongoDB souffre des problèmes à l'origine de la conception des filtres : pas de sémantique formelle, pas de schéma, pas de typage statique. D'autre part, son langage de requête déclaratif rend peu évidente la possibilité d'un encodage vers les filtres : des constructions concises et courantes dans le langage de requête MongoDB ont une sémantique ambiguë qui dépend du document en cours d'examen. L'ambiguité peut grandir exponentiellement avec la longueur de la requête et il est probable que les programmeurs MongoDB soit *a)* ne considèrent qu'un petit sous-ensemble des sens possibles des requêtes qu'ils écrivent, soit *b)* se reposent implicitement sur les données pour qu'elles restreignent le sens qui sera donné à leurs requêtes (et qu'elles continuent de le faire plus tard). Cette présence de motifs déclaratifs dans MongoDB nous permet d'explorer une nouvelle facette des filtres : à ce jour, le seul encodage d'un langage NoSQL vers les filtres est celui de Jaql ([4]), un langage bien plus orienté vers l'itération que vers la déclaration de transformations.

Concernant son absence de typage, un encodage vers les types et les filtres décrits plus haut serait utile pour MongoDB. Étant donné une ou plusieurs requêtes, le type généré offrirait un "schéma" approximatif de ce que la collection devrait contenir pour que la recherche soit un succès, et l'inférence de type appliquée aux filtres approximerait l'état de la collection après exécution d'une mise à jour. Il permettrait aussi d'offrir des avertissements et des erreurs, par exemple des avertissements concernant l'ambiguïté possible de la requête et des erreurs dans le cas de suppositions contradictoires sur les données présentes dans la collection ; par exemple, il est très simple dans MongoDB d'exprimer à la fois une recherche dans un champ appelé "0" et une recherche dans le premier élément d'un tableau. Notre inférence de type peut détecter l'ambiguïté ou, grâce à un contexte donné par d'autres parties d'une requête, résoudre l'ambiguïté.

Il faut cependant noter que les collections MongoDB sont hétérogènes : une information sur un élément d'une collection ne donne *aucune* information certaine sur les autres membres de cette collection. Notre travail repose sur le fait qu'en pratique, les gens continuent de maintenir des données relativement homogènes groupées par collection.

[1] Nous ignorons ici la présence d'index qui permettent d'optimiser ces situations et de les traiter en temps constant.

[2] En effet, nous verrons plus loin que les types que nous utilisons ne sont pas polymorphes. Ce choix permet aux filtres d'être polymorphes malgré le monomorphisme des types ; une version polymorphe a ensuite été développée en [5].

Contributions Notre principale contribution est la formulation d'un encodage d'un sous-ensemble non trivial de MongoDB vers les filtres et l'implémentation de cet encodage. L'analyse statique de requêtes MongoDB n'a à notre connaissance jamais été entrepris, ce malgré l'intérêt pour les programmeurs, en termes de sécurité et de vitesse de développement, de diminuer le nombre d'erreurs ne pouvant être découvertes qu'à l'éxécution. Sans la base théorique offerte par les filtres et les types proposés dans [4], ce travail aurait nécessité la construction d'un typeur *ad-hoc* dans un environnement dépourvu d'informations de types. Ceci, ainsi que la faible popularité de l'analyse statique chez les programmeurs MongoDB, explique peut-être pourquoi le travail présenté ici n'a pas été fait plus tôt.

Aperçu Nous présentons d'abord les filtres, leur sémantique, les types qu'ils manipulent et la relation de sous-typage dans la section 2 . Puis nous présentons MongoDB, sa syntaxe et ses particularités dans la section 3 et proposons enfin un encodage défini par induction sur la syntaxe de MongoDB vers les filtres dans la section 4 . La difficulté de l'encodage, par rapport au travail effectué sur Jaql dans [4], vient de l'écart conceptuel entre d'un côté les requêtes MongoDB, qui déclarent des transformations de données de façon *globale*, et de l'autre les filtres, qui effectuent ces transformations de façon *locale*.

2 . Les filtres et leurs types

Cette section présente de façon succinte les filtres développés dans [4] ; notre contribution commence dans la section 3 . Rappelons que les filtres effectuent des transformations sur des valeurs appartenant aux types que nous introduisons ci-dessous. La traduction de requêtes MongoDB vers les filtres et de valeurs MongoDB vers les types permettra d'appliquer l'inférence de type dont disposent les filtres aux requêtes MongoDB.

2 .1. Types

Un type est décrit par la syntaxe suivante :

Types t	$::=$	b (basique)	$\mid$	$t\&t$	(intersection)
	$\mid$	v (singleton)	$\mid$	$\neg t$	(negation)
	$\mid$	(t,t) (produit)	$\mid$	`empty`	(vide)
	$\mid$	$\{\ell : t, \ldots, \ell : t\}$ (*records* fermé)	$\mid$	`any`	(tout)
	$\mid$	$\{\ell : t, \ldots, \ell : t\ ,..\}$ (*records* ouvert)	$\mid$	$\mu T.t$	(type récursif)
	$\mid$	$t\|t$ (union)	$\mid$	T	(variable de récursion)

La sémantique d'un type est un ensemble de valeurs: un type de base b est un ensemble de valeurs atomiques prédéfini, le type singleton v est le type qui ne contient que la valeur v. Le type produit (t_1,t_2) contient toutes les paires dont le premier membre est de type t_1 et le second de type t_2. Le type *record* fermé {a:`int`, b:`int`} contient toutes les valeurs de *records* qui ont exactement deux champs a et b avec des valeurs entières, tandis que le type *record ouvert* {a: `int`, b:`int`, **..**} contient les valeurs de *records* qui ont *au moins* deux champs a et b avec des valeurs entières. Les connecteurs logiques correspondent aux opérations ensemblistes, par exemple $\neg t$ contient toutes les valeurs qui n'ont pas le type t. Les types spéciaux `empty` et `any` représentent respectivement le type inhabité et le type de toutes les valeurs. Enfin, l'opérateur $\mu T.t$ lie la variable T dans t à t lui-même.

On a vu que les types étaient interprétés comme des ensembles de valeurs ; cette sémantique définit une relation de sous typage : le type t_1 est un sous-type de t_2 (noté $t_1 \leq t_2$) si et seulement si l'ensemble des valeurs dénoté par t_1 est contenu dans celui dénoté par t_2. Par exemple, $(\texttt{null},\{a : \texttt{int}, b : \texttt{any}\}) \leq (\texttt{int}|\texttt{null},\{a : \texttt{int},..\})$. La définition formelle de la relation de sous-typage et de la procédure de décision pour la relation de sous-typage peut être trouvée en [3].

2 .2. Filtres

Les filtres peuvent être vus comme des transducteurs (automates d'arbres avec sortie), capables de capturer des variables, mais exprimés dans une syntaxe simple à manipuler pour le programmeur fonctionnel ; ils reproduisent la structure de leur entrée mais peuvent capturer des variables. Par exemple, le filtre $(2, x \Rightarrow x)$ (où le pattern matching $p \Rightarrow f$ lie plus fortement que le filtre paire (f,f)) échoue sur toute entrée autre qu'une paire. Sinon, il renvoie une paire dont le premier élément est 2 et le second est le second élément de son entrée. Les seuls filtres qui ignorent leur entrée sont les appels récursifs (de la forme Xa) et les expressions (e).

Un filtre est un terme généré par la grammaire suivante.

Filtres f	::=	e	(expression)	**Expression** e	::=	c	(constantes)
	|	$p \Rightarrow f$	(pattern)		|	x	(variables)
	|	(f,f)	(produit)		|	(e,e)	(paires)
	|	$\{\ell : f, \ldots, \ell : f\ ..\}$	(record)		|	$\{e : e, \ldots, e : e\}$	(*records*)
	|	$f \mid f$	(union)		|	$e + e$	(concaténation de *records*)
	|	$\mu X.f$	(récursion)		|	$e \setminus \ell$	(suppression de champ)
	|	$X\ a$	(appel récursif)		|	$\mathbf{op}(e, \ldots, e)$	(opérateurs prédéfinis)
	|	$f ; f$	(composition)		|	$f\ e$	(application de filtres)

Arguments a ::= $x \mid c \mid (a,a) \mid \{\ell : a, \ldots, \ell : a\}$

Patterns p	::=	t	(type)	|	$\{\ell : p, \ldots, \ell : p\ ..\}$	record ouvert	
	|	(p,p)	(paire)	|	$p \mid p$	ou/union	
	|	$\{\ell : p, \ldots, \ell : p\}$	record fermé	|	$p \& p$	et/intersection	

2 .2.1. Sémantique opérationnelle

(expr) $\dfrac{}{\delta;\gamma \vdash_{eval} e(v) \rightsquigarrow r}$ $r = \mathtt{eval}(\gamma, e)$

(prod) $\dfrac{\delta;\gamma \vdash_{eval} f_1(v_1) \rightsquigarrow r_1 \quad \delta;\gamma \vdash_{eval} f_2(v_2) \rightsquigarrow r_2}{\delta;\gamma \vdash_{eval} (f_1,f_2)(v_1,v_2) \rightsquigarrow (r_1,r_2)}$ si $r_1 \neq \Omega$ et $r_2 \neq \Omega$

(patt) $\dfrac{\delta;\gamma\ ,\ v/p \vdash_{eval} f(v) \rightsquigarrow r}{\delta;\gamma \vdash_{eval} (p \Rightarrow f)(v) \rightsquigarrow r}$ si $v/p \neq \Omega$

(comp) $\dfrac{\delta;\gamma \vdash_{eval} f_1(v) \rightsquigarrow r_1 \quad \delta;\gamma \vdash_{eval} f_2(r_1) \rightsquigarrow r_2}{\delta;\gamma \vdash_{eval} (f_1;f_2)(v) \rightsquigarrow r_2}$ si $r_1 \neq \Omega$

(union1) $\dfrac{\delta;\gamma \vdash_{eval} f_1(v) \rightsquigarrow r_1}{\delta;\gamma \vdash_{eval} (f_1 \mid f_2)(v) \rightsquigarrow r_1}$ si $r_1 \neq \Omega$

(union2) $\dfrac{\delta;\gamma \vdash_{eval} f_1(v) \rightsquigarrow \Omega \quad \delta;\gamma \vdash_{eval} f_2(v) \rightsquigarrow r_2}{\delta;\gamma \vdash_{eval} (f_1 \mid f_2)(v) \rightsquigarrow r_2}$

(rec) $\dfrac{\delta, (X \mapsto f);\gamma \vdash_{eval} f(v) \rightsquigarrow r}{\delta;\gamma \vdash_{eval} (\mu X.f)(v) \rightsquigarrow r}$

(rec-call) $\dfrac{\delta;\gamma \vdash_{eval} (\delta(X))(a) \rightsquigarrow r}{\delta;\gamma \vdash_{eval} (Xa)(v) \rightsquigarrow r}$

(error) $\dfrac{}{\delta;\gamma \vdash_{eval} f(a) \rightsquigarrow \Omega}$ à défaut

(recd) $\dfrac{\delta;\gamma \vdash_{eval} f_1(v_1) \rightsquigarrow r_1 \quad \cdots \quad \delta;\gamma \vdash_{eval} f_n(v_n) \rightsquigarrow r_n}{\delta;\gamma \vdash_{eval} \{\ell_1 : f_1, \ldots, \ell_n : f_n\ ..\}(\{\ell_1 : v_1, \ldots, \ell_n : v_n, \ldots, \ell_{n+k} : v_{n+k}\}) \rightsquigarrow \{\ell_1 : r_1, \ldots, \ell_n : r_n, \ldots, \ell_{n+k} : v_{n+k}\}}$ si $\forall i, r_i \neq \Omega$

La sémantique opérationelle à grands pas (définie dans [4]) est donnée par les règles d'inférence pour des jugements de la forme $\delta;\gamma \vdash_{eval} f(a) \rightsquigarrow r$ et décrit comment l'évaluation de l'application d'un filtre f à un argument a dans un environnement γ donne un objet r, où r est soit une valeur soit Ω (**error**). La plupart des règles représentent l'argument par v au lieu de a : v est un argument fermé, c'est à dire sans variables. Ω représente l'échec de l'application d'un filtre et est levé soit parce qu'un filtre n'a pas la forme de son argument, soit parce qu'un pattern matching a échoué. δ contient les corps des filtres récursifs.

Le filtre expression e (**expr**) jette son entrée et évalue e dans l'environnement courant.

Le filtre produit (f_1,f_2) (**prod**) attend une paire (v_1, v_2) en entrée et renvoie la paire composée du résultat de l'application de f_1 à v_1 et de l'application de f_2 à v_2. En particulier, ce filtre est utile pour appliquer un *mapping* à une liste : le premier composant f_1 transforme l'élément courant et f_2 est appliqué au reste.

Le filtre *record* $\{\ell_1 : f_1, \ldots, \ell_n : f_n \ , ..\}$ (**recd**) s'attend à recevoir une valeur de type record avec *au moins* les champs $\ell_1, \ldots, \ell_n$ spécifiés dans son corps et associés aux valeurs $v_1, \ldots, v_n$. Il applique ses sous-filtres aux champs spécifiés (respectivement f_1 à v_1, ..., f_n à v_n) et laisse les autres champs de l'entrée inchangés. L'application échoue si l'entrée n'est pas un record, ou si le record n'a pas certains des champs spécifiés par le filtre, ou si un sous-filtre échoue.

Un pattern p peut être vu comme un type non-récursif muni de variables de capture. Après l'évaluation d'un pattern p dans un filtre $p \Rightarrow f$ (**patt**) appliqué à une valeur v, f est évalué dans un environnement enrichi des variables capturées par p, et dénoté par v/p. Par exemple, le filtre $(x,(y,z)) \Rightarrow y + x$ appliqué à la valeur (2,(1,5)) renvoie 3, résultat de l'expression $x + y$ où x est lié à 2, y est lié à 1, et z est lié à 5. Pour une définition formelle de la capture de variable d'une variable v par un pattern p, voir [4]. L'échec de la capture fait échouer le filtre.

Le filtre union $f_1|f_2$ (**union1**, **union2**) tente d'appliquer f_1 à son entrée et ne lui applique f_2 que si l'application de f_1 a échoué.

Le filtre composition $f_1;f_2$ (**comp**) appliqué à une valeur v applique f_2 au résultat de l'évaluation de f_1 appliqué à v.

Enfin, le corps f d'un filtre récursif $\mu X.f$ (**rec**) est lié dans δ à la variable de récursion X avant d'être appliqué à son entrée ; on peut rapprocher cette construction d'un `let rec` $X = f$, au sens où $\mu X.f$ lie la variable X dans f à f lui-même. Enfin, comme pour le filtre expression, un appel récursif $X\,a$ (**rec-call**) ignore son entrée et applique le filtre lié à la variable de récursion X à son argument a.

2 .3. Exemple de filtre

Considérons le filtre F suivant :

```
μX.‘nil ⇒ ‘nil
  | ("stop" ⇒ "resume",x ⇒ x)
  | ({id : x ,..},tl) ⇒ ( x, X tl )
  | (h,tl) ⇒ X tl
```

F opère sur une liste (encodée à la LISP: $(e_1, (\ldots (e_n, \text{`nil}) \ldots)) = [e_1 \ldots e_n]$). Il remplace les éléments qui sont un *record* avec un champ id par la valeur de ce champ, pose une balise "resume" avant de s'arrêter s'il rencontre une balise "stop", et enlève les autres éléments.

Appliqué à la valeur [{id: 2; name: "Harry"} false 3 "stop" {id: 3; name: "Sherlock}], F renvoie [2 "resume" {id: 3; name: "Sherlock }].

2 .4. Inférence de type

Nous ne détaillerons pas l'algorithme de typage introduit dans [4], mais donnerons un aperçu de son fonctionnement. L'idée est, muni d'un type d'entrée, de procéder par induction sur la structure d'un filtre et de vérifier à chaque étape que le filtre est compatible avec l'entrée, puis d'évaluer au niveau des types les transformations effectuées par le filtre. Par exemple, considérons le filtre F suivant : $(0,x) \Rightarrow x \ | \ (1,y) \Rightarrow \{\text{snd} : y\}$. F prend en entrée une paire (u, v). Si u vaut 0, F renvoie v. Sinon, F renvoie un *record* dont l'unique champ snd est associé à la valeur v. L'inférence de type peut statiquement observer que l'application du filtre à une valeur de type `((0|1),int) | (1,string)`, c'est à dire soit à une paire dont le premier élément est 0 ou 1 et le second élément est une valeur du type de base `int`, soit

à une paire dont le premier élément est 1 et le second élément est une valeur du type de base `string` renverra une valeur de type `int | {`snd:(`int | string`)`}`, c'est à dire soit une valeur de type `int`soit un record dont l'unique champ snd est une valeur de type `int`ou une valeur de type `string`.

L'inférence ne termine pas pour toute paire filtre-type, mais une première passe d'analyse statique (introduite dans [4], section **4.5**) garantit, en cas de réussite, que l'inférence terminera. Tous les filtres construits dans cet article satisfont ce critère de terminaison.

Nous avons amélioré les algorithmes d'inférence de type et d'analyse statique introduits dans [4]. En particulier, l'inférence de type sur les filtres *record* perdait toute information sur les champs d'une valeur non spécifiés par le filtre qui la prenait en argument. Il est en fait possible de garder trace de ces champs.

3 . MongoDB et JSON

MongoDB est une base de données NoSQL. Elle utilise un encodage efficace de JSON (BSON) pour stocker ses documents, n'a pas de schéma, autorise le stockage de sous-documents à des profondeurs arbitraires et donne la priorité au fonctionnement à grande échelle plutôt qu'à l'expressivité de son langage de requête (par exemple il n'y a pas de jointures). C'est une base de données assez typique parmi les bases NoSQL.

JSON (Javascript Object Notation) est un format d'échange de données basé sur deux structures : les tableaux et les *records* non ordonnés (des collections de paires clé-valeur) ainsi que sur des valeurs atomiques (chaînes de caractères, nombres, booléens, null). Sa grammaire est la suivante : $v ::= \{ k{:}v, ..., k{:}v \} \mid [v, ..., v] \mid c$, où v est une valeur, c est une valeur atomique (entiers, chaînes, ...), k est une clé, et où les clés sont des chaînes de caractères (on omettra les guillemets lorsque la chaîne ne contient pas de caractères spéciaux autres que $). Pour une définition formelle, voir `www.json.org`.

Extension JSON MongoDB utilise un JSON étendu: *a)* Les champs dans les *records* sont ordonnés. Nous traitons cet aspect globalement dans la section 4 .2. *b)* de nouveaux types atomiques sont ajoutés, comme Date( ) : voir la section 4 .2. Le langage de requête de MongoDB utilise lui aussi la syntaxe JSON, ce qui force l'existence d'un préfixe réservé $ pour distinguer, dans les requêtes, les valeurs (par exemple `{type: Array}`) des opérateurs (par exemple `{$type: Array}`). À titre d'exemple, une projection (sélection d'une partie de chaque résultat) est un *record* dont les champs sont des chemins vers des valeurs MongoDB et dont les valeurs sont soit 1 (inclusion) soit 0 (exclusion). De façon similaire, une recherche est un *record* dont les champs sont des chemins et les valeurs sont soit des opérateurs de requête spéciaux (préfixés par $) soit des valeurs MongoDB.

Exemple de collection Une base MongoDB est faite de collections, qui sont des listes hétérogènes de *records* JSON étendu ou "documents". Les valeurs JSON étendu incluent de nouveaux types atomiques (voir plus bas). Une collection MongoDB avec deux documents pourrait être :

```
{ _id: ObjectId("f28a1f9ec0d5"),
  address: "221B Baker Street",
  name: "Holmes",
  stats: [4,2.7,2]
},
```

```
{ _id: ObjectId("5210b2c9084f"),
  address: ["4 Privet Drive", "12, Grimmauld Place"],
  name: "Potter",
  info: {
    jobs: [{title: "Student", risk: "Low"},
           {title: "Wizard", risk: "High"}]
  },
  stats: {books: 7, readers: "7m", films: 7}
}
```

Requêtes Les requêtes MongoDB utilisent deux parmi trois composants : une partie recherche (Q), une partie mise à jour (U), et une partie projection (P). find(Q, P) sélectionne les documents qui correspondent à la recherche Q et renvoie une projection P des résultats, par inclusion ou par exclusion. update(Q, U) met à jour les documents qui correspondent à la recherche Q au moyen d'une mise à jour U.

Chemins Appliquée à la collection ci-dessus, la recherche find`({stats.films: 7},{})`

sélectionne le second document, car il possède un champ `stats` dont la valeur est un *record* muni d'un champ `films` dont la valeur est 7. La valeur renvoyée est le document lui-même : la projection vide `{}` inclue tous les champs. Cependant, des documents à la structure assez différente auraient pu être sélectionnés par le chemin `stats.films` : un *chemin* est une liste de champs de valeurs séparés par des points et représente un algorithme d'accès aux sous-champs d'un document. La forme et la sémantique d'un chemin dépendent du contexte où on le trouve dans une requête MongoDB.

Syntaxe des types Les types de données manipulés par MongoDB sont des *records* (ordonnés et extensibles) et des listes hétérogènes de valeurs ; ils peuvent être représentés par la syntaxe suivante, qui peut ensuite être encodée vers la syntaxe de types donnée plus haut. Notons que ces types n'existent pas du point de vue de MongoDB, mais ils caractérisent bien les valeurs que MongoDB manipule.

Types MongoDB t	::=	v (singleton)	|	`int` | `string` | `binary` | `ObjectID` | ... (base)
	|	$\{\ell : t, \ldots, \ell : t\}$ (records fermés)	|	`empty` | `any` | `null` (spécial)
	|	$\{\ell : t, \ldots, \ell : t$, `..`$\}$ (records ouverts)	|	$t \mid t$ (union)
	|	$[r]$ (séquences)	|	$t \backslash t$ (différence)

Regexp r ::= $\varepsilon \mid t \mid r* \mid r+ \mid r? \mid r\ r \mid r|r$

Les listes hétérogènes sont encodés dans nos types réguliers à la LISP. Par exemple, les listes non vides pouvant contenir des éléments de type `int` ou `char` peuvent être représentées soit par le type $((\texttt{int}|\texttt{char}), \mu T\ (\texttt{nil} \mid ((\texttt{int}|\texttt{char}), T)))$ soit par l'expression régulière `[(int|char)+]`. ε est le mot vide, le type séquence $[r]$ est l'ensemble de toutes les séquences dont le contenu est décrit par l'expression régulière r. `[char*]` dénote `string`, l'ensemble des chaînes de caractères.

3 .1. Syntaxe

Nous dérivons une grammaire pour les les recherches, mises à jour et projections à partir de la documentation en anglais en MongoDB. Il s'agit d'un sous-ensemble de MongoDB : en particulier, nous ne traitons pas les fonctions d'aggrégation de type `groupby`.

3 .1.1. Recherches

q ::= $\{cond_1, \ldots, cond_n\}$
mod ::= `$and` | `$or` | `$nor`
p ::= $\ell\ r$
r ::= $\varepsilon \mid .\ell\ r \mid .i\ r$
cmp ::= `$gt` | `$gte` | `$lt` | `$lte`
comparable ::= *numeric* | *timestamp* | *datetime*
cond ::=
| p: v
| p: $\{op_1, \ldots, op_n\}$
| *mod*: $[q_1, \ldots, q_n]$

op ::=
| `$all`: *array* **a** ou **est égal à toutes les valeurs de** *array*
| *cmp*: *comparable* satisfait la comparaison *cmp* avec *comparable*
| `$in`: *array* **présent** ou **a une valeur** dans *array*
| `$ne`: v **n'est pas** ou **n'a pas pas de valeur** égale à v
| `$nin`: *array* **n'est pas** ou **n'a pas pas d'élément** dans *array*
| `$not`: $\{op_1, \ldots, op_n\}$ ne satisfait ni op_1, ..., ni op_n
| `$where`: *expr* *expr* renvoie true
| `$elemMatch`: q a un élément qui satisfait q
| `$size`: i est un tableau de taille i
| `$exists`: *boolean* test d'existence
| `$mod`: $[numeric_1, numeric_2]$ mod $numeric_1 = numeric_2$
| `$type`: *type* **est** ou **a une valeur** de type *type*)

Une recherche Q est un *record* de conditions. Une condition est soit une contrainte concernant un chemin dans le document examiné soit un combinaison booléenne de conditions. Les contraintes incluent l'égalité, la différence, l'absence, la présence, la comparaison, le type ...

string, ℓ, i, *numeric*, *array* et v sont respectivement des chaînes, des entiers positifs, tous les nombres (voir la section 4 .2), des tableaux et des valeurs JSON, avec deux restrictions : *a)* la première lettre des

noms de champs dans les valeurs JSON n'est jamais `$`, et *b)* quand `$or` ou `$nor` sont utilisés le tableau de leur requête a au moins deux éléments.

3 .1.2. Mises à jour

Dans la suite p dénotera un chemin de *mise à jour* (par exemple `a.b.0.c.$.d`) tandis que les chemins de recherches définis dans la section 3 .1.1 seront dénotés par p^q lorsqu'une mise à jour U associée à une recherche Q aura besoin d'extraire un chemin de cette recherche.

update	::=	$\{u_1, \ldots, u_n\}$	| `$inc`	incrémente valeur
u	::=	$op : \{p_1 : v_1, \ldots, p_n : v_n\}$	| `$unset`	supprimer
p	::=	$\ell\ r$	| `$addToSet`	ajoute au tableau si absent
r	::=	$\varepsilon \mid .\ell\ r \mid .\$\ r \mid .i\ r$	| `$pop`	supprime 1ère ou dernière valeur
op	::=	`$set` modifie valeur	| `$push`	ajoute en fin du tableau
		| `$rename` déplace valeur	| `$pull`	supprime valeur du tableau

De plus, les mises à jour doivent respecter les contraintes suivantes :

1. Les chemins et valeurs sous `$rename` doivent être des étiquettes ℓ séparées par des points (pas de nombre i ni de `$`).
2. Les valeurs sous `$inc` appartiennent à *numeric* et
3. Les valeurs sous `$addToSet` sont des tableaux de valeurs.[3]

3 .1.3. Syntaxe de projection

Dans ce qui suit, p dénotera un chemin de *projection*, tandis que les chemins de recherche comme définis plus tôt seront dénotés par p^q.

Les projections MongoDB sont des *record* de la forme $\{p_1 : \alpha_1, \ldots, p_n : \alpha_n\}$, où α parcourt 0, 1 et des opérations de la forme `{$elemMatch: cond}`. 0 dénote l'exclusion, 1 dénote l'inclusion.

Une projection obéit aux contraintes suivantes[4] : *a)* les valeurs numériques valent soit toutes 0 soit toutes 1, *b)* seuls les chemins composés d'une unique étiquette ℓ peuvent avoir des `{$elemMatch: cond}` pour valeur, et 3) `$` et `{$elemMatch: cond}` ne peuvent pas apparaître dans la même projection.

3 .2. Exemples

3 .2.1. find(Q, P)

Considérons la requête :

```
find({address:  "12, Grimmauld Place",                    } Query
    "info.jobs":  {$elemMatch:  {title:  "Wizard"}}},     }
    {name:  1, "info.jobs.$":  1})                        } Projection
```

Elle renvoie :

```
{ _id:  ObjectId("5210b2c9084f"),
  name:  "Potter",
```

[3] La forme exacte des valeurs de `$addToSet` est soit v soit `{$each: [`$v_1, \ldots, v_n$`]}`. Nous transformons la première forme en `[v]` et la seconde en `[`$v_1, \ldots, v_n$`]`

[4] Nous ignorons pour l'instant un cas particulier de MongoDB : par défaut, une liste d'inclusion se voit automatiquement ajouter `_id: 1` avant exécution à moins que `_id: 0` ne soit déjà présent dans la liste.

```
  info: {
    jobs: [{title: "Wizard", risk: "High"}]
  }
}
```

La partie recherche (Query) restreint les documents renvoyés à ceux qui ont :

1. un champ `address` égal soit à `"12, Grimmauld Place"`, soit à un tableau[5] contenant `"12, Grimmauld Place"`
2. un champ `info` égal soit à une valeur v soit à un tableau contenant v, où v est un *record* avec un champ `jobs` égal à un tableau l contenant au moins un record avec
   (a) un champ `title` égal soit à `"Wizard"` soit à un tableau contenant `"Wizard"`
   (b) un champ `risk` égal soit à `"High"` soit à un tableau contenant `"High"`

La partie Projection renvoie (implicitement) le champ `_id`, le champ `name` et le tableau l réduit au premier *record* qui satisfait (a) et (b).

On peut remarquer que :

1. Spécifier une valeur correspond à demander cette valeur ou un tableau contenant cette valeur. À notre connaissance, la seule façon de demander une égalité stricte est d'utiliser l'opérateur `$where` et d'écrire du code Javascript[6]
2. Dans la plupart des contextes, tout point (`.`) dans un chemin $p'.p''$ où p'' n'est pas vide permet deux possibilités : le côté droit du chemin (p'') peut soit faire référence à un *record* avec un champ égal au premier élément de p'' soit à un tableau contenant ce record.
3. `$elemMatch` vérifie qu'au moins un élément dans un tableau satisfait plusieurs conditions à la fois. Il force implicitement que le dernier élément de son chemin associé soit un tableau.
4. L'élément spécial `$` dans une projection P fait référence à un élément trouvé dans un tableau lors de la recherche Q correspondante tel que ce tableau puisse être adressé par le chemin à gauche de `$`. Dans cet exemple, il fait référence au `job` qui satisfait les conditions de la clause `$elemMatch`.

3 .2.2. update(Q, U)

Considérons la mise à jour `update({$exists: {stats: true}},{$inc: { 'stats.2': 1 })`. Elle laisse la collection donnée en exemple dans l'état suivant :

```
{ _id: ObjectId("f28a1f9ec0d5"),
  ⋮
  name: "Holmes",
  stats: [4,2.7,3]
},
```

```
{ _id: ObjectId("5210b2c9084f"),
  ⋮
  name: "Potter",
  stats: {"2": 1, books: 7, readers: "7m", films: 7}
}
```

La partie recherche restreint les documents mis à jour à ceux munis d'un champ `stats`.

[5] Dans MongoDB, les listes sont appelées tableaux ; nous utiliserons ce terme pour les distinguer des listes dans notre calcul de filtres.

[6] `$where`, qui prend une chaîne de Javascript comme argument, dépasse le cadre de notre encodage. De plus, utiliser l'opérateur `$type` combiné à l'opérateur négation ne fonctionne pas comme solution partielle car MongoDB fait un cas particulier lorsque le type utilisé avec `$type` est `Array`: dans ce cas, MongoDB cherche à vérifier que les éléments de la valeur (qui doit être un tableau) sont eux-mêmes des tableaux.

Le partie mise à jour augmente (`$inc`) la valeur associée au chemin `stats.2` de 1. Pour le document `Holmes`, c'est le 3ème élément du tableau `stats`, mais pour le document `Potter`, `stats` est un *record*, donc MongoDB voit 2 comme un nom de champ (et pas comme un index de tableau), crée ce champ et exécute l'action par défaut pour `$inc`, qui consiste à créer une valeur égale à l'incrément spécifié.

Si un champ du chemin `stats.2` avait été absent, MongoDB aurait créé un objet pour compléter la partie manquante du chemin ; si le tableau `stats` avait été trop petit, MongoDB l'aurait rempli d'élements `null` jusqu'à ce que sa taille soit 2 et aurait ensuite assigné au 3ème élément la valeur 1.

Dans ce cas, nous avons fait le choix d'interdire l'utilisation de nombres comme noms de champs ; notre expérience nous montre que dans la plupart des cas, MongoDB manifeste ce comportement suite à une erreur du programmeur. Le filtre correspondant à cette mise à jour soulèverait une erreur s'il était appliqué au document `Potter`.

4 . Encodage vers les filtres

On traduit une recherche Q en un type $[\![Q]\!]$, une update U accompagnée d'une recherche Q en un filtre $[\![U]\!]_Q$, et une projection P accompagnée d'une recherche Q en un filtre $[\![P]\!]_Q$. Les traductions de chacune seront données dans les sections suivantes. Les formes générales des encodages sont les suivantse :

On traduit une recherche $\mathsf{find}(Q, P)$, par Filter($[\![Q]\!]\&x \Rightarrow x$); Transform($[\![P]\!]_Q$), et une mise à jour $\mathtt{update}(Q, U)$, par Filter($[\![Q]\!]\&x \Rightarrow x$); Transform($[\![U]\!]_Q$), où Filter et Transform sont définis ci-dessous.

Filter(f) renvoie la liste des valeurs renvoyées par f sur les éléments d'une liste. Les applications échouées de f sont ignorées.

$\mu X.\mathtt{nil} \Rightarrow \mathtt{nil}$
  | (f, Id)
  | $(h, t) \Rightarrow (h, X\ t)$

Transform(f) est comme **Filter** mais échoue si l'une des applications de f échoue.

$\mu X.\mathtt{nil} \Rightarrow \mathtt{nil}$
  | (_, h) $\Rightarrow (f, \mathrm{X\,h})$

Nous introduirons d'abord l'encodage d'une recherche vers un type (section 4 .1), puis passerons à l'encodage d'une mise à jour vers un filtre (section 4 .3), et d'une projection vers un filtre (section 4 .4).

4 .1. Recherche

On applique les règles de réécriture suivantes avant d'effectuer la traduction vers les filtres. Elles préservent la sémantique et rendent la recherche uniforme :

$$
\begin{array}{rcl}
p\mathtt{:}\ v & \Rightarrow & p\mathtt{:}\ \{\mathtt{\$in:}\ [v]\} \\
cond_1, \ldots, cond_n & \Rightarrow & \mathtt{\$and:}\ [cond_1, \ldots, cond_n] \\
p\mathtt{:}\ \{op_1, \ldots, op_n\} & \Rightarrow & \mathtt{\$and:}\ [p\mathtt{:}\ \mathrm{op}_1, \ldots, p\mathtt{:}\ \mathrm{op}_n] \\
p\mathtt{:}\ \{\mathtt{\$not:}\ \{op_1, \ldots, op_n\}\} & \Rightarrow & \mathtt{\$and:}\ [p\mathtt{:}\ \{\mathtt{\$not:}\ op_1\}, \ldots, p\mathtt{:}\ \{\mathtt{\$not:}\ op_n\}]
\end{array}
$$

La première réécriture est la plus notable et révèle que, du point de vue de MongoDB, rechercher "une valeur" et "une valeur dans un tableau" a le même sens. Les trois autres règles donnent une forme normale à l'expression d'une conjonction de conditions.

4 .1.1. Conditions, chemins et opérateurs

On convertit maintenant la recherche vers un type. Rappelons que q est une abbréviation de *query*, la forme générale d'une recherche MongoDB. `$nor` recherche les éléments qui ne satisfont aucun de ses conditions.

mod	$[\![mod]\!]$
`$and: [`$q_1, \dots, q_n$`]`	$\&_{i=1}^{n} [\![q_i]\!]$
`$or: [`$q_1, \dots, q_n$`]`	$\mid_{i=1}^{n} [\![q_i]\!]$
`$nor: [`$q_1, \dots, q_n$`]`	`any\` $\mid_{i=1}^{n} [\![q_i]\!]$

op	$[\![op]\!]$ *(root node)*	$[\![op]\!]_{\wr}$ *(child node)*
`$all: [`$v_1, \dots, v_n$`]`	Ω	$(\&_{i=1}^{n} [\![v_i]\!]) \mid (\&_{i=1}^{n}$ `[any*` $[\![v_i]\!]$ `any*])`
`$ne:` v	Ω	`any\[`$[\![v]\!]$`]`$^?$
`$nin: [`$v_1, \dots, v_n$`]`	Ω	`any\[`$\mid_{i=1}^{n} [\![v_i]\!]$`]`$^?$
`$size:` i	Ω	`[any`^{n}`]`
cmp`:` v	$[\![v]\!]$	`[`$[\![v]\!]$`]`$^?$
`$in: [`$v_1, \dots, v_n$`]`	$\mid_{i=1}^{n} [\![v_i]\!]$	`[`$\mid_{i=1}^{n} [\![v_i]\!]$`]`$^?$
`$elemMatch: {`*cond*`}`	$[\![cond]\!]$	`[any*` $[\![cond]\!]$ `any*]`
`$mod: [`n,m`]`	numeric	[numeric]$^?$
`$type: Array`	`[any*]`	`[any* [any*] any*]`
`$type:` *BSONType*	$[\![BSONType]\!]$	$[[\![BSONType]\!]]^?$ si *BSONType* $\neq$ `Array`
`$exists: true`	`any`	`any`

path	$[\![path]\!]$ *(root node)*	$[\![path]\!]_{\wr}$ *(child node)*
ℓ*.p*`:` *op*	$\nexists?_{\ell.p:op}$	Ω si *op* = `$exists: false`
ℓ*.p*`:` *op*	{ℓ`:` $[\![p:\ op]\!]_{\wr}$`; .. }`$\mid \nexists?_{\ell.p:op}$	$[\{\ell:\ [\![p:\ op]\!]_{\wr}; \dots\}]^?$ sinon
i.p`:` *op*	Ω	`[any`^{i} {ℓ`:` $[\![p:\ op]\!]_{\wr}$`; .. } any*]`
ε`:` *op*	$[\![op]\!]$	$[\![op]\!]_{\wr}$

où

$$\nexists?_{p:op} = \begin{cases} \{..\} \setminus [\![p: \{\$exists:true\}]\!]_{\wr} & \text{si } op = \$in:[v_1, \dots, v_n] \text{ et } null \in \{v_1,...,v_n\}, \\ & \text{ou } op = \$nin:[v_1, \dots, v_n] \text{ et } null \notin \{v_1,...,v_n\}, \\ & \text{ou } op = \$ne:v \text{ et } v \neq null \\ & \text{ou } op = \$exists:false, \\ empty & \text{sinon} \end{cases}$$

Nous rappelons qu'une query q peut être une condition logique *mod:* `[...]` ou une paire *p:op* ; les traductions récursives de la première section (*mod*) peuvent donc soit partir vers la troisième section (*path*) soit rester dans la première.

[t]$^?$ = t | `[any*` t `any*]`. Ce sucre syntaxique explicite l'ambiguïté omniprésente entre *égalité* et *présence dans une liste* : [t]$^?$ signifie "soit une valeur de type t, soit une liste qui contient une valeur de type t".

`$and` et `$or` sont convertis respectivement en une intersection et une union de types, tandis que `$nor` est satisfait par les données qui ne satisfont aucune de ses conditions.

Seule la colonne droite de la partie *op* est utilisée dans cette section. La partie gauche, qui correspond à la recherche d'un élément satisfaisant *op* dans un tableau, est utilisé par l'encodage des updates.

`$all` est satisfait soit par une valeur égale à toutes celles spécifiées en argument (donc toutes les valeurs données en argument doivent être égales), soit à un tableau contenant toutes les valeurs spécifiées (sans préférence d'ordre).

`$elemMatch` force la valeur à être un tableau, d'où l'absence de $^?$. `$type` compare la valeur à un

type de données BSON (convertis en types réguliers dans la section 4 .2), ou, si la valeur est un tableau, requiert que le tableau contienne au moins une valeur du type spécifié. Le cas particulier `Array` n'autorise que le second cas.

Les traductions `$ne`, `$nin`, `$size`, *cmp*, `$in`, `$mod`, `$exists` reproduisent de façon simple les descriptions en français indiquées lors de la description de la syntaxe.

$\nexists?_{p:op}$ est utilisé par les opérateurs qui autorisent l'absence de valeur. En particulier, $\nexists?_{p:op}$ autorise l'absence d'une valeur si `null` est une valeur autorisée (`$in`), si elle n'est pas une valeur interdite (`$nin`, `$ne`), ou si *op* est l'opérateur d'absence (`$exists:false`). `null` correspond donc à la fois à la valeur `null` et à l'absence de valeur — il faut utiliser `$exists:false` pour n'autoriser que l'absence de valeur, et `$type` pour n'autoriser que la valeur `null`.

4 .2. Encodage des valeurs et types BSON

Puisque nos types sont des ensembles de valeurs, le type $[\![v]\!]$ d'une valeur v est simplement le singleton contenant cette valeur.

BSON (à la fois une extension de JSON et un encodage efficace de celui-ci) définit des types supplémentaires que nous encodons aussi.

- Min key, max key et null, qui contiennent chacun un unique atome frais, par exemple `null` pour null.
- Le type numeric utilisé lors de la conversion est l'union des doubles, des entiers 64 bits et des entiers 32 bits. Afin de conserver à la fois leur valeur et leur type, ils sont convertis en une paire (atome, valeur) où le premier élément est un atome frais qui indique leur type : `int32`, `int64` ou `double`.
- Les types binary data, ID, regexp, javascript, symbol et timestamp sont encodés de la même façon : le premier élément est un atome frais (`atom`, `binary`, ...) qui décrit le type et le second est un encodage réversible de la valeur (nous ne nous intéressons pas à cet encodage car nos filtres n'inspectent jamais cette valeur)[7].
- Les tableaux sont encodés comme des listes, les chaînes comme des listes de caractères ; puisque BSON n'a pas le type caractère, l'ensemble des chaînes BSON encodées est disjoint de l'ensemble des tableaux BSON encodés.
- Les booléens sont encodés par les atomes true et false.
- L'ordre est significatif dans les *record* BSON, c'est-à-dire : `{ premier: 1, second: 2 }` $\neq$ `{ second: 2, premier: 1 }`. Nous encodons les records BSON comme des paires :

  $\texttt{\{ } a_1 \texttt{: } v_1 \texttt{, ..., } a_n \texttt{: } v_n \texttt{\}} \quad \Rightarrow \quad ([a_1, ..., a_n],\{a_1: v_1, ..., a_n: v_n\})$

  Les changements à apporter à notre encodage sont alors triviaux mais très lourds, nous les omettons donc partout sauf ici : les valeurs et les patterns de *record* fermés sont encodés comme ci-dessus, les patterns de records ouverts deviennent des patterns de paires dont le premier membre est `[any*]`, et les filtres de records deviennent des filtres de paires dont le premier membre est le filtre Id.

  MongoDB réordonne les *record* pendant ses updates et laisse ce réordonnancement explicitement *non* défini. Nous faisons donc de même pour la concaténation des records et la suppression de champ : le premier membre du résultat de l'opération est simplement la liste des clés du second membre, dans n'importe quel ordre.

[7] Ce qui signifie que nos filtres ne reproduisent la sémantique de MongoDB que dans les types, pas dans l'exécution. On pourrait vouloir créer une version de l'encodage plus fine qui simule l'exécution exactement.

Ceci signifie que toute condition de recherche de la forme $p:\ r$ où r est un *record* échouera sur tout record r' dont les champs et les valeurs sont identiques mais dont l'ordonnancement des champs est différent. Ceci pose particulièrement problème dans le cas où r' avait été inséré avec l'ordonnancement de r mais qu'une modification de la valeur de l'un de ses champs a déclenché le mécanisme de réordonnancement de MongoDB et l'a transformé en r'.

Enfin, l'importance de l'ordre des champs sur des *record* signifie aussi que MongoDB ne peut ni *a)* rechercher un record fermé sans qu'un ordre soit spécifié, ni *b)* rechercher un record ouvert tout en spécifiant un ordre.

4 .3. Mise à jour

Nous appliquons la règle de réécriture suivante avant de convertir la mise à jour en filtre afin que l'ordre chemin-opération soit le même pour les recherches et les mises à jour :

$$op\ :\ \{p_1 : v_1, \ldots, p_n : v_n\} \quad \Rightarrow \quad p_1 : \{op\ :\ v_1\ \}, \ldots, p_n : \{op\ :\ v_n\ \}$$

Étant donnée une recherche Q, nous transformons une mise à jour U en un filtre. Par souci de clarté, nous n'écrivons pas $[\![U]\!]_Q$ mais $[\![U]\!]$. D'autre part, la conversion d'une paire chemin-opérateur $p:op$ en type de *recherche* sera indiqué par $[\![p:op]\!]^q$.

$$
\begin{array}{rcll}
[\![p_1 : \{op\ :\ v_1\ \}, \ldots, p_n : \{op\ :\ v_n\ \}]\!] & = & [\![p_1 : \{op\ :\ v_1\ \}]\!]\ ;\ \ldots\ ;\ [\![p_n : \{op\ :\ v_n\ \}]\!] & \\
[\![p : \{\texttt{\$set} : v\}]\!] & = & [\![\varepsilon\,|\,p]\!]_{x \Rightarrow v} & \\
[\![p : \{\texttt{\$inc} : v\}]\!] & = & [\![\varepsilon\,|\,p]\!]_{x \Rightarrow x+v} & \\
[\![p_1 : \{\texttt{\$rename} : p_2\}]\!] & = & \mathrm{Rename}(p_1, p_2) & \\
[\![p.\ell : \{\texttt{\$unset} : v\}]\!] & = & [\![\varepsilon\,|\,p]\!]_{x\&\{\ell\,:\,\mathrm{any}\,,..\} \Rightarrow x\setminus\ell} & \\
[\![p.a : \{\texttt{\$unset} : v\}]\!] & = & [\![\varepsilon\,|\,p.a]\!]_{x \Rightarrow \texttt{null}} & \text{avec } a ::= \texttt{\$} \mid i \\
[\![p : \{\texttt{\$addToSet} : [v_1, \ldots, v_n]\}]\!] & = & [\![\varepsilon\,|\,p]\!]_{\mathrm{AtS}(v_1)\ ;\ \ldots\ ;\ \mathrm{AtS}(v_n)} & \\
[\![p'\,|\,\ell.p]\!]_f & = & \{\ell :\ [\![p'.\ell\,|\,p]\!]_f\ ,..\} & \\
[\![p'\,|\,i.p]\!]_f & = & \mathrm{Position}_i([\![p'.i\,|\,p]\!]_f) & \\
[\![p'\,|\,\varepsilon]\!]_f & = & f & \\
[\![p'\,|\,\$.p]\!]_f & = & \Omega & \text{si } \mathrm{RQ}(p') \neq p^q : op \\
 & = & \mathrm{Position}_i([\![p'^q :\ op]\!]_{\hat{\varepsilon}}^{\,q} \Rightarrow [\![p'\,|\,p]\!]_f) & \text{si } p^q = i.p'^q \\
 & = & \mathrm{First}([\![p^q\colon op]\!]^q \Rightarrow [\![p'\,|\,p]\!]_f) & \text{sinon}
\end{array}
$$

où

$$
\begin{array}{rcll}
\mathrm{Position}_i(f) & = & (\mathrm{Id}_1, (\ldots (\mathrm{Id}_i, f, \mathrm{List}) \ldots)) & \\
\mathrm{List} & = & \mu X.\ \texttt{`nil} \Rightarrow \texttt{`nil} \mid (h,t) \Rightarrow (h, X\ t) & X \text{ frais} \\
\mathrm{Id} & = & x \Rightarrow x & \\
\mathrm{First(f)} & = & \mu X.\ (f, \mathrm{Id}) \mid (h,t) \Rightarrow (h, X\ t) & \\
\mathrm{AtS}(v) & = & \mu X.\ \texttt{`nil} \Rightarrow (v, \texttt{`nil}) & \\
 & & \quad \mid (v \& x \Rightarrow x\ , \mathrm{Id}) & \\
 & & \quad \mid (x,t) \Rightarrow (x, X\ t) & X \text{ frais} \\
\mathrm{Rename}(a_1.\ldots.a_n,\ b_1.\ldots.b_m) & = & \mathrm{Capture} \Rightarrow (\mathrm{Eliminate}\ ;\ \mathrm{Replace}_1) &
\end{array}
$$

Le filtre Rename est fait des 4 définitions suivantes, qui dépendent de $a_1.\ldots.a_n$ et $b_1.\ldots.b_m$.

$$
\begin{array}{rcll}
\text{Capture} & = & \{a_1 : \{ \ldots \{a_n : y; ..\} \ldots\}; ..\} & \text{(pattern)} \\
\text{Eliminate} & = & \{a_1 : \{ \ldots \{a_{n-1} : (x \Rightarrow x \backslash a_n); ..\} \ldots\}; ..\} & \text{(filter)} \\
\text{Replace}_i & = & \{b_i : \text{Replace}_{i+1}; ..\} \,|\, (\{..\} \& x \Rightarrow x + \text{Create}_i) & \forall i \leq m \text{(filter)} \\
\text{Replace}_{m+1} & = & y & \\
\text{Create}_i & = & \{b_i : \{ \ldots \{b_m : y\} \ldots\}\} & \forall i \leq m \text{(expression)}
\end{array}
$$

Initialisation La première section initialise l'encodage en séparant les opérations de mise à jour en plusieurs filtres composés et en donnant un filtre spécifique à chaque opération. Par exemple, `$addToSet` est associé au filtre AtS qui ajoute une valeur à une liste si cette valeur n'était pas dans la liste.

Descente La seconde partie construit des filtres imbriqués. Le chemin entier doit être gardé en mémoire à cause de l'opérateur positionnel `$` : si on le rencontre, tout le chemin jusqu'à `$` est utilisé par RQ sur la recherche Q associée à la mise à jour courante. Si un chemin est renvoyé par RQ, il est utilisé pour construire un filtre Position ou un filtre Find qui trouvera l'élément correspondant et lui appliquera le filtre de la mise à jour.

Le filtre de position $\text{Position}_i(f)$ applique le filtre f au $i$ème élément d'une liste ; notez qu'il ne remplit *pas* la liste avec des valeurs `null` comme MongoDB le fait lorsqu'il rencontre un tableau avec trop peu d'éléments. Il semble plus judicieux de lever une erreur lorsqu'on s'essaie à ce type de mise à jour.

Le filtre $\text{Find}(f)$ continue d'appliquer f à tous les éléments d'une liste jusqu'à ce que f n'échoue pas.

Opérateur positionnel `$` Étant donnée une recherche Q et un chemin de mise à jour `<path 1>.$.<path 2>`, `$` fait référence à l'index du premier élément du tableau situé à `<path 1>` qui a été trouvé grâce à une condition présente dans Q de la forme `<path 1>.<path 3>`: *op*. Notez que `<path 2>` et `<path 3>` peuvent être vides.

Par exemple, si Q = `{items: {$elemMatch: {color: "blue"}}}` et U = `{$set: {"items.$.price": 19}}`, `$` sera remplacé par l'index du premier élément du tableau `items` avec un champ `color` de valeur `["blue"]`[?].

L'algorithme RQ permet de retrouver `<path 3>`:*op* ; RQ(p) est le suffixe du chemin le plus à droite dans Q ayant pour préfixe p et n'étant ni sous un `$or` ni sous un `$nor`. Pour le trouver, il suffit de traverser en profondeur l'AST de la recherche en partant de la droite, de sauter tout noeud *mod* autre que `$and`, et de renvoyer Ω si aucun chemin adéquat n'a été trouvé. Par exemple, si Q = `{'a.b.c':op}`, RQ(a.$) = b.c : *op*.

Rename Étant donné deux chemins p et p', Rename capture la valeur v à laquelle p fait référence, l'enlève, et soit *a)* remplace la valeur v' référencée par p' par v soit *b)* construit un *record* tel que p' référence v. Sa particularité vient du fait que le filtre doit être capable de construire toute partie manquante de p'.

4 .4. Filtre de projection

Pour correspondre au comportement de MongoDB, on traite la projection comme suit : pour i allant de 1 à n (c'est-à-dire de gauche à droite), s'il y a une paire $p_j : \alpha_j$ telle que $i < j$, et que $p_i \sqsubset p_j$, on supprime simultanément toutes les occurrences de $p_i : \alpha_i$. Une inclusion est donc ignorée s'il en existe une plus spécifique à sa droite.

Puis on sépare la projection en deux : une *match liste M* composée de tous les p :*cond* tels qu'il y a un p :`{$elemMatch:`*cond*`}` dans la projection et une *inclusion/exclusion list L* des chemins $p_1, ..., p_n$ restants (associés soit tous à 0 soit tous à 1).

4 .4.1. Projection par inclusion

Étant donnée une recherche Q, une *inclusion list* L et une *match list* M, nous souhaitons traiter les chemins comme un arbre, c'est-à-dire traiter les descendants de préfixes identiques au même moment. Pour ce faire, nous définissons la fonction t pour obtenir les descendants :

$t(p) = \{\, a \mid \exists p' \in L \cup M \;\; p.a \sqsubseteq p' \,\}$ (p un chemin, a parcourt les étiquettes, les entiers positifs et \$)

Avec les définitions suivantes, notre filtre d'inclusion est $f_{\text{incl}} = [\![\varepsilon]\!]$.

$$
\begin{array}{rcll}
[\![p]\!] & = & [\![p \,|\, \$]\!]_{\text{Id}} & \text{si } \$ \in t(p) \\
[\![p]\!] & = & \text{Match}_p \;\mathbf{;}\; (\; \text{Proj}_p \;\mathbf{|}\; \text{Filter}_{\text{Proj}_p} \;) & \text{sinon} \\
\\
\text{Match}_\ell & = & \text{Find}([\![cond]\!]^q \Rightarrow \text{Id}) & \text{si } \ell : cond \in M \\
\text{Match}_\ell & = & \text{Id} & \text{sinon} \\
\text{Get}_{p,a_1,\ldots a_k} & = & x \Rightarrow x & \text{si } p \in L \cup M \\
\text{Get}_{p,a_1,\ldots a_k} & = & x \Rightarrow \{a_1 : \square, \ldots, a_k : \square\} \oplus_\square x & \text{sinon}
\end{array}
$$

$$
\begin{array}{rcll}
\text{Proj}_p & = & \{a_1 : [\![p.a_1]\!] \;, \mathbf{..}\} \;\mathbf{|}\; \{\mathbf{..}\} \Rightarrow x \backslash a_1 \;\mathbf{;} \\
 & & \ldots \;\mathbf{;} \\
 & & \{a_k : [\![p.a_k]\!] \;, \mathbf{..}\} \;\mathbf{|}\; \{\mathbf{..}\} \Rightarrow x \backslash a_k \;\mathbf{;} \\
 & & \text{Get}_{p,a_1,\ldots,a_k} \\
 & & \text{si } t(p) = \{a_1, \ldots, a_k\}, k > 0 \\
\text{Proj}_p & = & \text{Id} & \text{si } t(p) = \varnothing
\end{array}
$$

où $\square$ est un atome frais.

Ignorer les autres descendants dans $[\![p]\!]$ quand $\$ \in t'(p)$ revient à supprimer tous les frères et tous les descendants de \$, ce qui est désirable car MongoDB ignore ces inclusions lorsqu'un chemin contient \$. Notez que $[\![\text{p} \,|\, \$]\!]$ est la traduction vers filtre d'une mise à jour (voir section 4 .3).

La partie importante est le premier cas de Proj_p. Pour que chaque champ contienne le chemin actuel, il projette récursivement sur ses sous-champs ou le supprime si la projection échoue.[8] Si le chemin courant est spécifiquement inclus, le filtre renvoie le *record* modifié (premier cas de Get). Sinon, le filtre agrège les valeurs des champs à inclure dans un nouveau record : pour tout champ ℓ:v d'un *record* r, $r \oplus_\square r'$ remplace sa valeur par celle du champ $\ell : v'$ de r' si $v = \square$ et la laisse inchangée sinon. En particulier, si $v = \square$ mais que le champ ℓ de r' est indéfini, le champ ℓ de r devient indéfini. Pour une discussion plus détaillée de la sémantique des records dans le calcul des filtres, voir [4].

4 .4.2. Projection par exclusion

Étant donnée une requête Q, une *liste d'exclusion* L et une *match list* M, nous souhaitons exclure toutes les valeurs référencées par L et restreindre toutes celles référencées par M. Nous supprimons d'abord toutes les paires $\ell : cond$ dans M pour lesquelles il y a un chemin p' dans L tel que p' commence par ℓ. L est $p_1, ..., p_n$ et M est maintenant $\ell_1 : cond_1, ..., \ell_k : cond_k$.

Notre filtre d'exclusion est $f_{\text{excl}} = f_1 \;\mathbf{;}\; \ldots \;\mathbf{;}\; f_n \;\mathbf{;}\; m_1 \;\mathbf{;}\; \ldots \;\mathbf{;}\; m_k$ où $f_i = [\![p' : \texttt{\{\$unset: true\}}]\!]$ | Id et $m_i = \{\ell_i : \text{Match}_{\ell_i} \;, \mathbf{..}\} \;\mathbf{|}\; \text{Id}$.

5 . Conclusion

L'encodage présenté ici ouvre à la possibilité *a)* d'une référence formelle pour implémenteurs souhaitant comprendre la sémantique de MongoDB, *b)* d'un outil de typage qui, allié à des outils tels que MongoDB-Schema, offrirait au programmeur

- Un point de vue abstrait de ses données extrait de l'ensemble des requêtes fournies.

[8] Il faut donc noter que le comportement de MongoDB en cas d'échec de la projection diffère selon le contexte : si un `$elemMatch` échoue, ou si on tente de projeter sur une valeur qui n'est pas un *record*, la clé à laquelle cette valeur est associée est supprimée ; mais si une projection échoue sur un record, la clé correspondante *n'est pas* supprimée Exemples : *a)* Projeter `{a:[{b:false}]}` sur `{a:{$elemMatch:{b:true}}}` renvoie `{}`, *b)* Projeter `{a:{b:true}}` sur `{'a.b.x':1}` renvoie `{a:{}}`, et *c)* Projeter `{a:{b:{c:true}}}` sur `{'a.b.x':1}` renvoie `{a:{b:{}}}`.

- Un moyen d'éliminer statiquement certaines classes d'erreurs de ses requêtes, par exemple en constatant un avertissement lorsqu'il utilise l'opérateur *p: v*, généralement utilisé pour les tests d'égalité, sur une collection où *p* peut faire référence à un tableau.
- Une garantie que ses données ne s'écartent pas d'un schéma fourni suite à une mise à jour, par exemple qu'un certain champ `amount` n'est jamais effacé par un `$rename` ou un `$unset`.

Cet encodage a été implémenté sur une branche du langage ℂDuce [1] (qui implémente les filtres et les primitives de manipulation ensembliste de nos types) et dérive une paire type-filtre à partir d'une requête MongoDB. Le type représente le schéma des documents à laquelle la requête s'appliquera et le filtre peut être appliqué nativement à tout type d'entrée afin d'observer une version abstraite des transformations que la requête effectuera sur une collection réelle.

Les parties non traitées de MongoDB sont : les opérateurs `$where`, `$setOnInsert`, `$bit`, `$regex` et `$slice`, les opérateurs géospatiaux, et les requêtes d'aggrégation de type map-reduce.

Travaux futurs : MongoDB n'a pas de sémantique formelle grâce à laquelle vérifier l'encodage. La prochaine étape du travail consiste donc à générer un jeu de tests exhaustif qui permettra de valider l'encodage par le comportement observé de MongoDB lors des tests. D'autre part, les filtres possèdent des opérateurs `groupby` et `orderby` grâce auxquels nous comptons encoder les requêtes d'aggrégation de MongoDB. Sur un plan plus pratique, nous comptons produire un site internet destiné aux programmeurs MongoDB et dont l'interface permettrait de valider la cohérence de requêtes ou d'observer leur effet sur une base munie d'un schéma donné. Enfin, nous projetons de produire un encodage d'une recherche *Q* vers un filtre (en plus d'un type) afin de compléter la formalisation de la sémantique de MongoDB — nous nous sommes concentrés sur la traduction vers un type car cela nous permet d'offrir au programmeur MongoDB une première approximation des données de sa collection.

Bibliographie

[1] ℂDuce. http://cduce.org. Accédé le 10/10/2013.

[2] MongoDB. http://www.mongodb.org. Accédé le 10/10/2013.

[3] Alain FRISCH : *Théorie, Conception et Réalisation d'un langage de programmation adapté à XML*. Thèse de doctorat, Université Paris-Diderot 7, 2004.

[4] V. BENZAKEN, G. CASTAGNA, K. NGUYỄN et J. SIMÉON : Static and dynamic semantics of NoSQL languages. *In POPL'13, 40th ACM Symposium on Principles of Programming Languages*, p. 101–113, 2013.

[5] G. CASTAGNA, K. NGUYỄN, Z. XU, H. IM, S. LENGLET et L. PADOVANI : Polymorphic functions with set-theoretic types. part 1: Syntax, semantics, and evaluation. jun 2013.

[6] Kim NGUYỄN : *Langage de Combinateurs pour XML: Conception, Typage, Réalisation*. Thèse de doctorat, Université Paris-Sud 11, 2008.

Réseaux de Kahn à rafales et horloges entières

Adrien Guatto[1,3] & Louis Mandel[2,3]

1: École normale supérieure
2: Collège de France
3: INRIA Paris-Rocquencourt

Résumé

Les langages flot de données synchrones à la Lustre proposent un formalisme équationnel de haut niveau dédié à la conception et l'implantation de systèmes temps réel. Ils sont traditionnellement restreints aux systèmes critiques ne nécessitant pas de calcul intensif; en particulier, le code impératif généré ne contient pas naturellement de boucles.

Lucy-n est une variante récente de Lustre plus adaptée aux traitements multimédias. Dans cet article, nous proposons une extension de la sémantique de Lucy-n où les flots transportent des rafales de valeurs plutôt que de simples scalaires, ainsi qu'un système de types qui caractérise la taille de ces rafales. L'ambition est d'adapter les techniques de génération de code usuelles pour produire des boucles imbriquées.

1. Modèle n-synchrone et horloges entières

Le modèle n-synchrone [5] a été inventé pour la programmation d'applications vidéos. Il est né d'une collaboration entre des industriels spécialistes de ce domaine et des universitaires experts en langages synchrones [1] à partir du constat suivant.

D'une part, les industriels programmaient leurs applications sous la forme de réseaux de Kahn [6]. Ce modèle décrit les applications comme des réseaux de processus communiquant des flots de données par des buffers. Il a l'avantage de concilier concurrence et déterminisme mais peut nécessiter l'utilisation de buffers de taille infinie. Les ingénieurs devaient donc vérifier manuellement que leurs programmes pouvaient s'exécuter en mémoire bornée et calculer les tailles de buffers.

D'autre part, les universitaires avaient montré [3] que les programmes écrits dans les langages synchrones flots de données décrivaient un sous-ensemble des réseaux de Kahn. Un des avantages de ces langages est qu'ils disposent d'une analyse statique, appelée *calcul d'horloge*, qui borne la taille des buffers du réseau sous-jacent. Cependant, cette analyse impose une taille de un pour tous les buffers !

Le modèle n-synchrone relâche cette dernière contrainte, mais conserve le calcul d'horloge qui permet de garantir l'exécution en mémoire bornée. De plus, il fournit un cadre pour calculer automatiquement la taille des buffers. Le langage Lucy-n [9] a été proposé pour programmer en utilisant ce modèle. C'est un langage synchrone flot de données auquel un opérateur `buffer` a été ajouté.

Afin d'assurer l'exécution des programme en mémoire bornée, le modèle synchrone introduit une notion de temps logique global qui est défini comme une succession d'instants discrets. Ainsi, à chaque flot de valeurs, on peut associer une horloge indiquant la présence où l'absence de données à chaque instant. Le calcul d'horloge est un système de types qui assure que lors de l'application d'un opérateur, les arguments sont disponibles et que, lorsque le résultat est produit, le consommateur est prêt à l'utiliser.

En Lucy-n, comme dans les autres langages synchrones, les horloges sont des flots booléens indiquant à chaque instant s'il y a une valeur ou non dans un flot donné. S'il souhaite transporter

simultanément plusieurs valeurs dans le même flot, le programmeur doit utiliser des structures de données composites comme les tableaux. Cette approche lui laisse le choix de la représentation de ses données, mais aussi la responsabilité de les convertir d'un format en un autre. Par exemple, une image peut être vue comme un flot de pixels ou un flot de lignes de pixels. Pour passer d'une représentation à l'autre, le programmeur doit écrire des convertisseurs série/parallèle.

Dans cet article, nous proposons une approche alternative. L'objectif est de laisser le compilateur introduire automatiquement ce code de conversion. Pour cela, plutôt que de manipuler explicitement des tableaux, le langage autorise le présence simultanée de plusieurs valeurs dans chaque flot. Techniquement, comme imaginé dès les débuts du n-synchrone [4], les horloges sont étendues avec des entiers. Ceux-ci correspondent aux nombres de valeurs présentes à chaque instant dans un flot.

La section 2 présente cette idée intuitivement à travers un exemple. Le langage est ensuite défini formellement section 3. Les propriétés algébriques des horloges entières et le système de types sont décrits section 4. Enfin, la section 5 discute d'un prototype de compilateur.

2. Des horloges booléennes aux horloges entières

Le programme Lucy-n suivant est une fonction, ou *nœud*, qui calcule le produit scalaire de deux flots de vecteurs de taille trois arrivant composante par composante. En pratique, il additionne, trois éléments par trois éléments, le produit de ses deux flots d'entrée :

```
let node dotp_3 (x, y) = o where
  rec p = x * y
  and p0 = p when (true false false)
  and p1 = p when (false true false)
  and p2 = p when (false false true)
  and o = buffer p0 + buffer p1 + buffer p2
```

x	3	2	5	4	2	3	...
y	2	4	6	5	3	7	...
p	6	8	30	20	6	21	...
p0	6			20			...
p1		8			6		...
p2			30			21	...
o			44			47	...

La plupart des opérateurs du langage sont déjà présents dans les langages à la ML : définitions locales mutuellement récursives (`where rec`), fonctions, paires. La particularité du langage est qu'il manipule des flots de valeurs. Ainsi, les opérateurs combinatoires comme `*` s'exécutent point à point : ici, la i-ème valeur du flot `p` est égale au produit des i-èmes valeurs de `x` et `y` ($p_i = x_i \times y_i$).

L'opérateur binaire e `when` ce, spécifique aux langages flots de données synchrones, reçoit un flot de valeurs de type quelconque e et un flot de booléens ce, et conserve les valeurs de e uniquement lorsque ce est vraie. En Lucy-n, ce est restreint à un mot binaire ultimement périodique : `(true false false)` représente la répétition infinie du motif entre parenthèses. Dans le nœud `dotp_3`, l'opérateur `when` permet définir trois sous-flots `p0`, `p1` et `p2` constitués respectivement des premières, deuxièmes et troisièmes valeurs modulo trois du flot `p`.

Enfin, l'opérateur `buffer` est spécifique au langage. Il signifie que l'utilisateur souhaite l'introduction d'un buffer borné au point indiqué. Ces buffers préservent l'ordre de messages et sont sans duplication ni perte.

Calcul d'horloge Le caractère synchrone du langage vient de l'hypothèse suivante : lorsqu'un opérateur du langage consomme ou produit des données, il le fait *instantanément*. Par exemple, un opérateur arithmétique comme `*` produit une valeur en sortie pour une valeur sur chacune de ses entrées, et impose donc que ses entrées et sorties soient présentes exactement aux mêmes instants. Ces instants sont caractérisés par les types d'horloges des flots, et chaque programme Lucy-n induit donc un ensemble de contraintes sur les types d'horloges de ses sous-expressions.

Beaucoup de ces contraintes sont des égalités, à l'image de celles issues de l'opérateur `*`. En revanche, la relation entre le type d'horloges d'entrée et le type d'horloges de sortie de `buffer` est une

relation de sous-typage notée <:. Informellement, cette relation garantit que la communication par buffer fini est correcte, c'est-à-dire qu'il n'y a pas de risque de lire dans le buffer alors qu'il est vide ou d'y écrire alors qu'il est plein.

Le nœud `dotp_3` ci-dessus nous permet d'illustrer ce fonctionnement. Appelons respectivement α_x, α_y, α_p et α_o les types d'horloges de `x`, `y`, `p` et `o`. La définition de `p` utilisant un opérateur arithmétique, elle implique que les types d'horloges de `x`, `y` et `p` sont égaux.

L'opérateur de filtrage e `when` ce produit une sortie présente sur l'horloge de e filtrée par la condition booléenne ce. Ici, `p0`, `p1` et `p2` ont donc pour types d'horloges respectifs α_p `on` (1 0 0), α_p `on` (0 1 0) et α_p `on` (0 0 1), `on` dénotant la composition de deux *types d'horloges* et 1 et 0 la présence et l'absence de valeurs. Intuitivement, l'opérateur *on* compose deux *horloges* en parcourant celle de droite au rythme de celle de gauche. Par exemple, (1 0) *on* (1 0 0) se calcule de la façon suivante :

(1 0)	1	0	1	0	1	0	...
(1 0 0)	1		0		0		...
(1 0) *on* (1 0 0)	1	0	0	0	0	0	...

On peut constater ici que si l'entrée d'un `when` arrive un instant sur deux et que la condition conserve une entrée sur trois, la sortie est présente un instant sur six.

Enfin, les sorties des trois buffers utilisés pour définir `o` sont combinées via `+` : leurs types doivent donc être égaux à celui de `o` et respectivement sous-type de ceux de `p0`, `p1` et `p2`. Le système de contraintes final induit par ce nœud est donc :

$$\left\{\begin{array}{rcl} \alpha_x & = & \alpha_y \\ \alpha_y & = & \alpha_p \\ \alpha_p \texttt{ on } (1\ 0\ 0) & <: & \alpha_o \\ \alpha_p \texttt{ on } (0\ 1\ 0) & <: & \alpha_o \\ \alpha_p \texttt{ on } (0\ 0\ 1) & <: & \alpha_o \end{array}\right\}$$

Horloges binaires Interrogé au sujet de ce programme, le compilateur Lucy-n standard calcule sa signature d'horloge et la taille de chacun des trois buffers :

```
val dotp_3 : (int * int) -> int
val dotp_3 :: forall 'a. ('a * 'a) -> 'a on (0 0 1)
Buffer line 6, characters 10-19: size = 1
Buffer line 6, characters 22-31: size = 1
Buffer line 6, characters 34-43: size = 0
```

La première ligne consiste en un schéma de type de données standard à la ML. La seconde décrit les relations temporelles entre les entrées et sorties. Cette signature correspond à la solution $\alpha_x = \alpha_y = \alpha_p = \alpha$ et $\alpha_o = \alpha$ `on` (0 0 1), avec α une variable d'horloge fraîche[1]. On vérifie la satisfaction du système précédent en y remplaçant les inconnues par leurs valeurs :

$$\left\{\begin{array}{rcl} \alpha & = & \alpha \\ \alpha & = & \alpha \\ \alpha \texttt{ on } (1\ 0\ 0) & <: & \alpha \texttt{ on } (0\ 0\ 1) \\ \alpha \texttt{ on } (0\ 1\ 0) & <: & \alpha \texttt{ on } (0\ 0\ 1) \\ \alpha \texttt{ on } (0\ 0\ 1) & <: & \alpha \texttt{ on } (0\ 0\ 1) \end{array}\right\}$$

Ce système de contraintes est toujours satisfait car quelle que soit la valeur de α, les deux premières égalités sont toujours satisfaites. Par ailleurs, il a été montré dans [10] que $w_1 <: w_2$

1. Le compilateur utilise `'a` comme syntaxe concrète compatible ASCII pour α.

```
type dotp_3_mem =
  { b0: int; b1: int; tick: int; }
let dotp_3_step x y mem =
  let p = x * y in
  if mem.tick mod 3 = 0
  then push mem.b0 p;
  if mem.tick mod 3 = 1
  then push mem.b1 p;
  let o =
    if mem.tick mod 3 = 2
    then pop mem.b0 + pop mem.b1 + p
    else (- 1)
  in
  mem.tick <- mem.tick + 1;
  o
```

(a) Horloges binaires, $\alpha_o = \alpha$ on (0 0 1)

```
type dotp_3_mem = unit
let dotp_3_step x y mem =
  let p = Array.create 3 0 in
  for i = 0 to 2 do
    p.(i) <- x.(i) * y.(i)
  end;
  p.(0) + p.(1) + p.(2)
```

(b) Horloges entières, $\alpha_o = \alpha$

FIGURE 1 – Codes générés pour le nœud Lucy-n `dotp_3`

implique w on $w_1 <: w$ on w_2. Donc pour vérifier la satisfaction des trois contraintes de sous-typage, il suffit de vérifier les trois contraintes suivantes : $(1\ 0\ 0) <: (0\ 0\ 1)$, $(0\ 1\ 0) <: (0\ 0\ 1)$ et $(0\ 0\ 1) <: (0\ 0\ 1)$. Il s'agit de contraintes sur des mots ultimement périodiques, appelées contraintes d'*adaptabilité*. Intuitivement, elles sont satisfaites si les deux mots ont la même proportion de 1 et de 0 et si pour tout indice donné, il y a toujours eu plus d'occurrences de 1 dans le mot de gauche que dans celui de droite. Ces deux conditions sont satisfaites dans notre exemple.

Intéressons nous maintenant à la production de code impératif. En Lucy-n comme en Lustre, le but est d'obtenir une fonction de transition produisant, à partir de l'état courant du programme, la valeur du flot au prochain instant ainsi que le nouvel état. Les flots y sont donc représentés comme des scalaires. La compilation modulaire de tels langages est décrite dans l'article [2].

Le code OCaml de la figure 1(a) pourrait être généré à partir de `dotp_3` et de la solution décrite. Le nœud est compilé vers une fonction de transition recevant ses entrées ainsi que son état courant. L'état est une structure à trois champs mutables : deux correspondent aux buffers de taille non nulle, le troisième est un entier indiquant l'instant courant. La fonction met à jour les buffers et calcule la sortie en fonction de l'horloge de ceux-ci. L'horloge est calculée via le compteur `mem.tick`. Remarquons que la valeur -1 définissant `o` lorsque son horloge vaut zéro (au premier et deuxième instant) ne sera jamais utilisée par un appelant si celui-ci respecte la signature d'horloge de `dotp_3`.

Vers les horloges entières Le nœud `dotp_3` calcule le produit scalaire de flots de vecteurs de taille trois reçus linéairement composante par composante. Avec les types d'horloges calculés précédemment, les deux buffers de taille non nulle permettent, tous les trois instants, de rendre disponible simultanément les produits des deux composantes arrivées aux deux instants précédents.

La sémantique de `when` nous assure que la sortie de `dotp_3` sera produite trois fois plus lentement que l'entrée. En Lucy-n, elle sera absente deux instant sur trois par rapport au rythme d'arrivée des entrées. Le code généré doit donc être activé trois fois pour obtenir un résultat. Une alternative serait de recevoir deux tableaux de taille trois pour produire un scalaire à chaque appel, de manière à obtenir le code OCaml de la figure 1(b).

Si les codes de la figure 1 implémentent la même fonctionnalité, ils ont des caractéristiques différentes en terme d'usage des ressources. Le premier est économe en espace, le second en temps ; il paraît difficile de décréter l'un supérieur à l'autre *a priori*. Le programmeur aimerait guider le compilateur pour générer ces deux implémentations à partir du même source initial et d'éventuelles annotations.

Produire une sortie à chaque instant exige de recevoir trois composantes sur `x` et `y` par instant. Nous appellerons un tel paquet de données une *rafale*. Pour prendre en compte les rafales dans le calcul d'horloge, nous étendons les horloges avec des valeurs entières. Ainsi, on peut proposer la solution suivante au système de contraintes de `dotp_3` : $\alpha_x = \alpha_y = \alpha_p = \alpha$ `on` (3) et $\alpha_o = \alpha$ `on` (1). Ceci produit le système de contraintes suivant :

$$\left\{\begin{array}{rcl} \alpha \texttt{ on } (3) & \texttt{=} & \alpha \texttt{ on } (3) \\ \alpha \texttt{ on } (3) & \texttt{=} & \alpha \texttt{ on } (3) \\ \alpha \texttt{ on } (3) \texttt{ on } (1\ 0\ 0) & \texttt{<:} & \alpha \texttt{ on } (1) \\ \alpha \texttt{ on } (3) \texttt{ on } (0\ 1\ 0) & \texttt{<:} & \alpha \texttt{ on } (1) \\ \alpha \texttt{ on } (3) \texttt{ on } (0\ 0\ 1) & \texttt{<:} & \alpha \texttt{ on } (1) \end{array}\right\}$$

Le calcul du *on* s'étend simplement au cas des mots entiers. Le mot de gauche indique de nombre de valeurs à lire le mot de droite :

(3)	3	3	3	...
(1 0 0)	1 0 0	1 0 0	1 0 0	...
(3) *on* (1 0 0)	1	1	1	...

De façon similaire, (3) *on* (0 1 0) = (1) et (3) *on* (0 0 1) = (1). Donc les trois contraintes de sous-typage deviennent α `on` (1) **<:** α `on` (1) et sont donc satisfaites.

Cette solution est découverte par notre prototype de compilateur capable d'inférer des types d'horloges entières lorsqu'on autorise la formation de rafales :

```
val dotp_3
  : (int * int) -> int
  :: ('a on (3) * 'a on (3)) -> 'a
```

À partir de cette signature, il serait possible de générer un code similaire à celui déjà présenté figure 1(b). La fonction OCaml `dotp_3_step` manipule des tableaux et produit une sortie valide par instant. Les types d'horloges à gauche et à droite de chaque contrainte de sous-typage étant les mêmes, les buffers sont de taille nulle. Ainsi, le type `dotp_3_mem` ne contient pas d'information.

Notations Dans ce qui suit, étant donné un ensemble X, nous notons X^* (resp. X^ω) les séquences finies (resp. infinies) d'éléments de X, et ε la séquence finie vide. La longueur d'une séquence finie w est dénotée par $|w|$. Pour une séquence finie de booléens cs, on note $|cs|_{\mathbf{1}}$ (resp. $|cs|_{\mathbf{0}}$) le nombre de booléens vrais (resp. faux) dans cs. On utilise $x[i]$ avec $i \in \mathbb{N}$ pour accéder au i-ème élément de la séquence x, et $x[i,j]$ avec $0 \leq i \leq j$ pour accéder à la sous-séquence de x débutant à l'indice i et terminant à l'indice j. Par abus de notation, $x[i,\dots]$ désigne le flot x privé de ses $i-1$ premiers éléments. On distingue la séquence finie x de $[x]$, cette dernière représentant la séquence (de séquences) de longueur un contenant comme unique élément x

Un élément appartenant à l'ensemble $X^\infty \triangleq X^* \cup X^\omega$ est appelé *flot* (d'éléments de X). On note $x.y$ la concaténation de deux flots, avec $x.y = x$ si x est infini. L'ordre préfixe sur X^∞ est défini par $x \sqsubseteq y$ s'il existe un flot z tel que $x.z = y$. L'ensemble X^∞ ordonné par $\sqsubseteq$ admet ε comme plus petit élément et forme ainsi un ordre partiel complet. Par convention, $u, v, \dots$ désignent des flots finis et $w, w', \dots$ des flots infinis.

Étant donné un ordre partiel complet X et une fonction $f : X \to X$ continue, nous notons $\mathit{fix}_X(f)$ son plus petit point fixe.

Enfin, les opérateurs en gras (**<:**, `on`, etc) sont des opérateurs sur les types et leur version en italique est leur interprétation sur les mots infinis ($<:$, *on*, etc).

3. Lucy-n et rafales

Si le nœud `dotp_3` précédent admet plusieurs types d'horloges induisant des comportements temporels différents, son action sur les éléments d'un flot est, elle, unique. Pour formaliser cette intuition, nous dotons le langage de deux sémantiques. La première, dite *de Kahn* est atemporelle, indépendante du calcul d'horloge, et manipule des flots de valeurs scalaires. La seconde, dite *synchrone*, manipule des flots de rafales dont la taille est déterminée par ses types d'horloges. Nous montrerons ensuite que ces sémantiques coïncident pour les programmes bien typés.

Nous décrivons les sémantiques d'une variante idéalisée de Lucy-n baptisée Core Lucy-n, essentiellement identique au langage décrit dans la thèse de Plateau [10]. Nous choisissons par souci de simplicité d'évacuer la question orthogonale des types de données. Dans le reste de l'article, les termes *types* et *typage* désigneront respectivement les types d'horloges et le typage d'horloge. On appelle les valeurs de base des *scalaires*. On note S l'ensemble des *scalaires*, celui-ci devant contenir les booléens, les entiers et être fermé par produit.

Le langage On suppose que l'ensemble N_X des noms de variables $x, y, \ldots$, l'ensemble N_F des noms de nœuds $f, g, \ldots$ et l'ensemble N_α des noms de variables de type $\alpha_1, \alpha_2, \ldots$ sont disjoints. Core Lucy-n est un langage à base d'expressions qui sont décrites par la grammaire suivante.

Exp	$\ni$	e	$::=$	c^{st}	constante littérale
			$\mid$	x	variable
			$\mid$	$f^{st}\ e$	application
			$\mid$	(e, e)	couple
			$\mid$	`fst` e $\mid$ `snd` e	projections
			$\mid$	e `where rec` $x = e$	définition locale
			$\mid$	e `when` ce	filtrage
			$\mid$	`merge` $ce\ e\ e$	fusion
			$\mid$	`sync` e	synchronisation
			$\mid$	$(\texttt{buffer}\ e)^{st}$	mise en mémoire
CExp	$\ni$	ce	$::=$	$b^*(b^+)^{st}$	condition booléenne ultimement périodique

Certaines expressions sont annotées avec un type st. Les types sont formés de variables et de conditions de type. Celles-ci peuvent être entières. La syntaxe des types et conditions de type est :

STy	$\ni$	st	$::=$	α	variable d'horloge
			$\mid$	st `on` stc	horloge composée
STyCond	$\ni$	stc	$::=$	$b^*(b^+) = b$	condition booléenne
			$\mid$	$i^*(i^+)$	mot d'entiers ultimement périodique

Enfin, un programme Core Lucy-n complet est une suite de définitions :

Def	$\ni$	*def*	$::=$	`let node` $f^\alpha\ x$ `=` e	nœud
			$\mid$	*def*; *def*	séquence de définitions

Chaque déclaration `let node` $f^\alpha\ x$ `=` e est annotée avec une variable de type α, quantifiée universellement [2]. Intuitivement, cette variable sera instanciée par l'horloge indiquant le lien avec l'horloge de l'appelant.

2. On se limite à une seule variable uniquement pour simplifier la présentation.

Sémantique de Kahn Les valeurs manipulées par la sémantique de Kahn sont des flots de scalaires. Ces valeurs obéissent à la grammaire suivante :

$$\begin{array}{rcll} V_K & \ni & v & ::= & S^\infty & \text{flot de scalaires} \\ & & & | & (v,v) & \text{couple de valeurs de Kahn} \end{array}$$

Pour toute expression e, sa sémantique de Kahn, notée $[\![e]\!]^K$, est une fonction des environnements de Kahn dans les valeurs de V_K. Un environnement de Kahn σ est la donnée d'une fonction $\sigma_v : N_X \to V_K$ et d'une fonction $\sigma_f : N_F \to (V_K \to V_K)$ envoyant les noms de variables et de nœuds dans leurs dénotations respectives.

Étudions la sémantique de chacune des expressions du langage. Les constantes donnent naissance à des flots infinis.

$$\begin{array}{rcl} [\![c^{st}]\!]^K(\sigma) & = & repeat^\#(c) \text{ avec} \\ & & repeat^\#(c) = c.repeat^\#(c) \end{array}$$

Leur sémantique n'utilise pas les annotations de type st car la sémantique de Kahn est atemporelle et ne nécessite donc pas d'horloges.

Les variables, applications, paires, projections et définitions locales récursives reçoivent leur sémantique habituelle.

$$\begin{array}{lcl} [\![x]\!]^K(\sigma) & = & \sigma_v(x) \\ [\![f^{st}\ e]\!]^K(\sigma) & = & (\sigma_f(f))\ ([\![e]\!]^K(\sigma)) \\ [\![(e_1,e_2)]\!]^K(\sigma) & = & ([\![e_1]\!]^K(\sigma), [\![e_2]\!]^K(\sigma)) \\ [\![\texttt{fst}\ e]\!]^K(\sigma) & = & v_1 \text{ avec } [\![e]\!]^K(\sigma) = (v_1,v_2) \\ [\![\texttt{snd}\ e]\!]^K(\sigma) & = & v_2 \text{ avec } [\![e]\!]^K(\sigma) = (v_1,v_2) \\ [\![e_1 \texttt{ where rec } x = e_2]\!]^K & = & [\![e_1]\!]^K(\sigma + [x \mapsto v]) \text{ avec} \\ & & v = fix_{V_K}(\lambda v.[\![e_2]\!]^K(\sigma + [x \mapsto v])) \end{array}$$

L'opérateur binaire de filtrage `when` reçoit un flot de scalaires et un flot booléen baptisé *condition*. Il filtre les valeurs du premier en fonction du second :

$$\begin{array}{rcll} [\![e \texttt{ when } ce]\!]^K(\sigma) & = & when^\#([\![e]\!]^K(\sigma), [\![ce]\!]^K) \text{ avec} & \\ & & when^\#(\varepsilon, \_) & = \varepsilon \\ & & when^\#(\_, \varepsilon) & = \varepsilon \\ & & when^\#(x.xs, t.cs) & = x.when^\#(xs, cs) \\ & & when^\#(x.xs, f.cs) & = when^\#(xs, cs) \\ & & when^\#(\_, f^\omega) & = \varepsilon \end{array}$$

L'opérateur `merge` fusionne deux flots en fonction d'une condition booléenne. Si la condition est vraie, il sélectionne le premier flot, sinon le second :

$$\begin{array}{rcll} [\![\texttt{merge } ce\ e_1\ e_2]\!]^K(\sigma) & = & merge^\#([\![ce]\!]^K, [\![e_1]\!]^K(\sigma), [\![e_2]\!]^K(\sigma)) \text{ avec} & \\ & & merge^\#(\varepsilon, \_, \_) & = \varepsilon \\ & & merge^\#(\_, \varepsilon, \_) & = \varepsilon \\ & & merge^\#(\_, \_, \varepsilon) & = \varepsilon \\ & & merge^\#(t.cs, x.xs, ys) & = x.merge^\#(cs, xs, ys) \\ & & merge^\#(f.cs, xs, y.ys) & = y.merge^\#(cs, xs, ys) \end{array}$$

L'opérateur `sync` transforme un couple de flots en flot de couples :

$$\begin{array}{rcll} [\![\texttt{sync } e]\!]^K(\sigma) & = & sync^\#([\![e]\!]^K(\sigma)) \text{ avec} & \\ & & sync^\#(\varepsilon, \_) & = \varepsilon \\ & & sync^\#(\_, \varepsilon) & = \varepsilon \\ & & sync^\#(x.xs, y.ys) & = (x,y).sync^\#(xs, ys) \end{array}$$

L'opérateur `buffer` correspond intuitivement à un décalage temporel fini entre les dates des entrées et des sorties. La sémantique de Kahn étant atemporelle, cet opérateur est l'identité.

$$[\![\texttt{buffer}\ e]\!]^K(\sigma) \quad = \quad [\![e]\!]^K(\sigma)$$

La sémantique des flots conditionnels ultimement périodiques est :

$$[\![u_b(v_b)^{st}]\!]^K \quad = \quad u_b.repeat^{\#}(v_b)$$

Enfin, on interprète un nœud par une fonction, et la sémantique de la composition de définitions est la composition des sémantiques de chacune.

$$\begin{array}{lcl} [\![\texttt{let node}\ f^{\alpha}\ x\ \texttt{=}\ e]\!]^K & = & \sigma + [f \mapsto \lambda v.([\![e]\!]^K(\sigma + [x \to v]))] \\ [\![def_1; def_2]\!]^K(\sigma) & = & [\![def_2]\!]^K([\![def_1]\!]^K(\sigma)) \end{array}$$

Cette sémantique ne manipule que des flots de scalaires. Le temps est absent, et il n'est donc pas possible de parler de simultanéité ou d'absence de valeur. On décrit donc maintenant une sémantique où chaque scalaire est calculé à une date donnée par rapport à un temps logique global.

Sémantique synchrone Les valeurs de base de cette sémantique sont des flots contenant des rafales. Ces rafales sont simplement des séquences finies de scalaires. La séquence vide représente l'absence de valeur :

$$\begin{array}{lcll} V_S \ni v & ::= & (S^*)^{\infty} & \text{flot de rafales} \\ & | & (v, v) & \text{couple de valeurs synchrones} \end{array}$$

Étant donné un tel flot de rafales xs, on définit son *horloge* comme le flot d'entiers $clock^{\#}(xs)$ où à tout rang i de xs, l'entier $(clock^{\#}(xs))[i]$ dénote la taille de la rafale $xs[i]$:

$$\begin{array}{lcl} clock^{\#} & : & (S^*)^{\infty} \to \mathbb{N}^{\omega} \\ clock^{\#}(\varepsilon) & = & repeat^{\#}(0) \\ clock^{\#}(x.xs) & = & |x|.clock^{\#}(xs) \end{array}$$

La fonction $pack^{\#}$ permet de construire des flots de rafales à partir d'une horloge et d'un flot de scalaires :

$$\begin{array}{lcl} pack^{\#} & : & \mathbb{N}^{\omega} \to S^{\infty} \to (S^*)^{\infty} \\ pack^{\#}(\_, \varepsilon) & = & \varepsilon \\ pack^{\#}(n.ck, xs) & = & [xs[1, n]].pack^{\#}(ck, xs[n+1, \omega]) \end{array}$$

Symétriquement, la fonction $unpack^{\#}$ aplatit les rafales pour récupérer le flot de scalaires sous-jacents :

$$\begin{array}{lcl} unpack^{\#} & : & (S^*)^{\infty} \to S^{\infty} \\ unpack^{\#}(\varepsilon) & = & \varepsilon \\ unpack^{\#}(x.xs) & = & x.unpack^{\#}(xs) \end{array}$$

Étudions maintenant la sémantique synchrone $[\![e]\!]^S$ des expressions. C'est une fonction des environnements synchrones dans les flots de rafales V_S. Comme pour un environnement de Kahn, un environnement synchrone σ spécifie pour chaque nom de variable x sa valeur $\sigma_v(x)$ et pour chaque nom de nœud g sa sémantique $\sigma_f(g)$. Mais un environnement synchrone spécifie également pour chaque variable de type α sa valeur $\sigma_{st}(\alpha)$ qui est une horloge $w \in \mathbb{N}^{\omega}$.

La sémantique synchrone des constantes, contrairement à la sémantique de Kahn, utilise l'annotation de type pour calculer l'horloge de la constante et crée les rafales avec l'opérateur $pack^{\#}$.

$$[\![c^{st}]\!]^S(\sigma) \quad = \quad pack^{\#}([\![st]\!]^S(\sigma), repeat^{\#}(c))$$

Cette définition nécessite donc l'interprétation des types $[\![st]\!]^S$ et des conditions entières $[\![stc]\!]^S$:

$$\begin{array}{lcl}
[\![\alpha]\!]^S(\sigma) & = & \sigma_{st}(\alpha) \\
[\![st \texttt{ on } stc]\!]^S(\sigma) & = & [\![st]\!]^S \; on \; [\![stc]\!]^S \\
\\
[\![u_i(v_i)]\!]^S & = & u_i.repeat^{\#}(v_i) \\
[\![u_b(v_b) = b]\!]^S & = & bool^{\#}(u_b.repeat^{\#}(v_b), b) \text{ avec} \\
 & & \quad bool^{\#}(x.cs, b) \;=\; (if\ x = b\ then\ 1\ else\ 0).bool^{\#}(cs, t) \\
\\
[\![u_b(v_b)^{st}]\!]^S(\sigma) & = & pack^{\#}([\![st]\!]^S(\sigma), u_b.repeat^{\#}(v_b))
\end{array}$$

L'intuition de l'opérateur *on* de composition d'horloge est la suivante : composer deux horloges w et w' consiste à exécuter w' au rythme de base défini par w. Quand w contient des entiers supérieurs à un, cela implique de fusionner le nombre d'activation correspondant dans w'. Fusionner des activations signifie additionner les entiers correspondants :

$$\begin{array}{rcl}
on & : & \mathbb{N}^\omega \to \mathbb{N}^\omega \to \mathbb{N}^\omega \\
(n.w)\ on\ w' & \triangleq & (\sum_{i=1}^{n} w'_i).(w\ on\ w'[n+1, \ldots])
\end{array}$$

Par exemple, $(2\ 0)\ on\ 0(1) = 1(0\ 2)$:

$$\begin{array}{r|cccccc}
(2\ 0) & 2 & 0 & 2 & 0 & 2 & \ldots \\
0(1) & 0\ 1 & & 1\ 1 & & 1\ 1 & \ldots \\
(2\ 0)\ on\ 0(1) & 1 & 0 & 2 & 0 & 2 & \ldots
\end{array}$$

Les opérateurs `when`, `merge` et `sync` sont appliqués point à point aux rafales :

$$\begin{array}{lcl}
[\![e \texttt{ when } ce]\!]^S(\sigma) & = & map_2^{\#}(when^{\#}, ([\![e]\!]^S(\sigma), [\![ce]\!]^S(\sigma))) \\
[\![\texttt{merge } ce\ e_1\ e_2]\!]^S(\sigma) & = & map_3^{\#}(merge^{\#}, ([\![ce]\!]^S(\sigma), [\![e_1]\!]^S(\sigma), [\![e_2]\!]^S(\sigma))) \\
[\![\texttt{sync } e]\!]^S(\sigma) & = & map_2^{\#}(sync^{\#}, [\![e]\!]^S(\sigma))
\end{array}$$

Les fonctions de la famille $map_k^{\#}$ utilisées ci-dessus appliquent une fonction d'arité k à k flots :

$$\begin{array}{lcl}
map_k^{\#} & : & ((S^*)^k \to S^*) \to ((S^*)^\infty)^k \to (S^*)^\infty \\
map_k^{\#}(f, (\ldots, \varepsilon, \ldots)) & = & \varepsilon \\
map_k^{\#}(f, (x_1.xs_1, \ldots, x_k.xs_k)) & = & f(x_1,\ \ldots\ x_k).map_k^{\#}(f, (xs_1, \ldots, xs_k))
\end{array}$$

La sémantique synchrone des buffers change l'horloge d'un flot. Pour cela, elle transforme le flot de rafales d'entrée en un flot de scalaires avec l'opérateur $unpack^{\#}$. Puis, comme pour les constantes, elle utilise l'annotation de type pour reconstruire un flot de rafales sur une nouvelle horloge.

$$\begin{array}{lcl}
[\![(\texttt{buffer } e)^{st}]\!]^S & = & buffer^{\#}([\![st]\!]^S(\sigma), [\![e]\!]^S(\sigma)) \\
 & & \text{avec } buffer^{\#}(w, xs) \;=\; pack^{\#}(w, unpack^{\#}(xs))
\end{array}$$

Pour les déclarations de nœuds, l'environnement des nœuds est enrichi avec les nouvelles définitions. Par rapport à la sémantique de Kahn, les fonctions prennent un argument supplémentaire qui est l'horloge de base du nœud. Celle-ci permet de calculer l'interprétation des types qui sont utilisés dans le corps du nœud.

$$\begin{array}{lcl}
[\![\texttt{let node } f^\alpha\ x \texttt{ = } e]\!]^S(\sigma) & = & \sigma + [f \mapsto \lambda w.v.([\![e]\!]^S(\sigma + [\alpha \mapsto w, x \mapsto v]))] \\
[\![def_1; def_2]\!]^S(\sigma) & = & [\![def_2]\!]^S([\![def_1]\!]^S(\sigma))
\end{array}$$

Lors de l'appel d'un nœud, il faut donc instancier cet argument avec l'interprétation de l'horloge de l'appel donnée par l'annotation de type.

$$[\![f^{st}\ e]\!]^S(\sigma) \;=\; (\sigma_f(f))\ ([\![st]\!]^S(\sigma))\ ([\![e]\!]^S(\sigma))$$

Enfin, la sémantique des variables, paires, projections et définitions récursives est identique au cas de la sémantique de Kahn.

4. Horloges entières

4.1. Système de types

Nous présentons d'abord quelques définitions et propriétés des horloges entières comme objets mathématiques [10] [3]. Nous définissions ensuite le système de types fondé sur cette algèbre d'horloge. Les propriétés seront implicitement utilisées par le système de types et justifieront sa correction.

Algèbre des horloges L'opérateur de composition *on* défini précédemment est associatif et admet un élément neutre et un absorbant :

Propriété 1. *Pour toute horloge w, w' et w'' :*
- w *on* $(1) = (1)$ *on* $w = w$
- w *on* $(0) = (0)$ *on* $w = (0)$
- $(w$ *on* $w')$ *on* $w'' = w$ *on* $(w'$ *on* $w'')$

La sémantique de l'opérateur `buffer` n'impose *a priori* aucune relation entre l'horloge de son entrée et l'horloge de sa sortie. Nous souhaitons toutefois rejeter certains types de comportements :
- on ne lit jamais dans un buffer vide ;
- le buffer est de capacité finie (et on n'écrit jamais dans un buffer plein).

Ces conditions peuvent être formalisées via la fonction de cumul d'une horloge w : $\mathcal{O}_w(i)$. Cette fonction fournit la quantité totale de données transportée par w du premier au i-ème instant. Elle est définie par :

$$\begin{array}{lcl} \mathcal{O} & : & \mathbb{N}^\omega \to \mathbb{N} \to \mathbb{N} \\ \mathcal{O}_w(0) & \triangleq & 0 \\ \mathcal{O}_{n.w}(1+i) & \triangleq & n + \mathcal{O}_w(i) \end{array}$$

La quantité de données contenue dans un buffer dont l'entrée (resp. la sortie) est horloge w (resp. w') à la fin de l'instant i peut maintenant être décrite comme $\mathcal{O}_w(i) - \mathcal{O}_{w'}(i)$. La première propriété de ne jamais lire dans un buffer vide peut être caractérisée par la relation de *précédence*, notée $\preceq$. La seconde propriété qui garantit que le nombre d'écritures et de lectures dans le buffer ne va pas diverger peut être caractérisée par la relation de *synchronisabilité*, notée $\bowtie$. La conjonction des deux relations forme la relation d'*adaptabilité* notée $<:$ et évoquée en section 2 :

$$\begin{array}{lcl} w \preceq w' & \triangleq & \forall i \in \mathbb{N}.\ \mathcal{O}_w(i) \geq \mathcal{O}_{w'}(i) \\ w \bowtie w' & \triangleq & \exists b \in \mathbb{N}.\ \forall i \in \mathbb{N}.\ -b \leq \mathcal{O}_w(i) - \mathcal{O}_{w'}(i) \leq b \\ w <: w' & \triangleq & w \preceq w' \wedge w \bowtie w' \end{array}$$

Une propriété importante de l'adaptabilité est sa préservation par changement d'horloge de base. C'est en effet pour cette raison que l'on peut instancier l'horloge de base d'un nœud par une valeur arbitraire.

Lemme 1. *Pour toutes horloges w et w' et à chaque instant i, on a $\mathcal{O}_{w \texttt{ on } w'}(i) = \mathcal{O}_{w'}(\mathcal{O}_w(i))$.*

Propriété 2. *Pour toutes horloges w, w' et w'', si $w <: w'$ alors w'' `on` $w <: w''$ `on` w'.*

Démonstration. Prouvons la précédence :

$$\begin{array}{lcll} w'' \texttt{ on } w \preceq w'' \texttt{ on } w' & \Leftrightarrow & \forall i.\ \mathcal{O}_{w'' \texttt{ on } w}(i) \geq \mathcal{O}_{w'' \texttt{ on } w'}(i) & \{ \text{ par définition de } \preceq \ \} \\ & \Leftrightarrow & \forall i.\ \mathcal{O}_w(\mathcal{O}_{w''}(i)) \geq \mathcal{O}_{w'}(\mathcal{O}_{w''}(i)) & \{ \text{ par le lemme 1 } \} \\ & \Leftrightarrow & \mathit{true} & \{ \text{ par } w \preceq w' \ \} \end{array}$$

Le raisonnement est identique pour la synchronisabilité. □

3. Une formalisation Coq de ces propriétés par Florence Plateau et Louis Mandel est disponible à l'adresse `http://lucy-n.org/coq/inw_prop.html`.

$$\textsc{Const}\ \frac{}{\Gamma \vdash c^{st} :: st} \qquad \textsc{Var}\ \frac{\Gamma(x) = st}{\Gamma \vdash x :: st} \qquad \textsc{App}\ \frac{\Gamma(f) = ty_{sch} \qquad Inst(ty_{sch},\ st'') = st, st' \qquad \Gamma \vdash e :: st}{\Gamma \vdash f^{st''}\ e :: st'}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash e_1 :: st_1 \qquad \Gamma \vdash e_2 :: st_2}{\Gamma \vdash (e_1, e_2) :: st_1 \times st_2} \qquad \textsc{Fst}\ \frac{\Gamma \vdash e :: st_1 \times st_2}{\Gamma \vdash \texttt{fst}\ e :: st_1} \qquad \textsc{Snd}\ \frac{\Gamma \vdash e :: st_1 \times st_2}{\Gamma \vdash \texttt{snd}\ e :: st_2}$$

$$\textsc{When}\ \frac{\Gamma \vdash e :: st}{\Gamma \vdash e\ \texttt{when}\ p^{st} :: st\ \texttt{on}\ (p = \texttt{true})} \qquad \textsc{Merge}\ \frac{\Gamma \vdash e_1 :: st\ \texttt{on}\ (p = \texttt{true}) \qquad \Gamma \vdash e_2 :: st\ \texttt{on}\ (p = \texttt{false})}{\Gamma \vdash \texttt{merge}\ p^{st}\ e_1\ e_2 :: st}$$

$$\textsc{Sync}\ \frac{\Gamma \vdash e :: st \times st}{\Gamma \vdash \texttt{sync}\ e :: st} \qquad \textsc{Buffer}\ \frac{\Gamma \vdash e :: st \qquad st <: st'}{\Gamma \vdash (\texttt{buffer}\ e)^{st'} :: st'}$$

$$\textsc{Where}\ \frac{\Gamma + [x \mapsto st] \vdash e_2 :: st \qquad \Gamma + [x \mapsto st] \vdash e_1 :: st'}{\Gamma \vdash e_1\ \texttt{where rec}\ x = e_2 :: st'}$$

FIGURE 2 – Calcul d'horloge pour Core Lucy-n.

Enfin, remarquons que les horloges ultimement périodiques sont des objets manipulables algorithmiquement.

Propriété 3. *La composition de deux horloges ultimement périodiques est calculable et ultimement périodique. L'adaptabilité de deux horloges ultimement périodiques est décidable.*

Munis de ces éléments mathématiques, on décrit maintenant les propriétés des types d'horloges et du calcul d'horloge, reliant horloges, types et programmes.

Types composés et calcul d'horloge Les types d'horloges simples dénotés par st caractérisent des flots de scalaires ou des flots de couples de scalaires. Le langage comprenant également des couples de flots dont les deux composantes peuvent être d'horloges différentes, nous introduisons les types d'horloges composés ty.

ty	::=		type d'horloges composé
	|	st	type d'horloges simple
	|	$ty \times ty$	type produit

Un nœud Core Lucy-n est polymorphe en ses horloges de base décrits par des schémas de types ty_{sch}. Ceux-ci ne lient ici qu'une seule variable, par simplicité.

$$ty_{sch} \quad ::= \quad \forall \alpha.\ ty \ \rightarrow\ ty \quad \text{schéma de type d'horloges composé}$$

Le jugement de typage $\Gamma \vdash e :: ty$ indique que dans le contexte de typage Γ, l'expression e a le type composé ty. Un environnement Γ associe un nom de variable à un type composé et un nom de nœud à un schéma de type.

La figure 2 présente les règles du calcul d'horloge pour les expressions. On retrouve pour les variables, paires, projections, déclarations et définitions composées les règles classiques de ML. Une

constante est typée selon son annotation (règle CONST). Le type d'horloges de e `when` ce est comme attendu celui de e composé avec ce (règle WHEN). La fusion de deux flots étant la contrepartie binaire du filtrage, la règle MERGE est symétrique. L'opérateur `sync` reçoit un couple de flots dont les rafales sont de longueur égale et produit un flot de couples par rafales de même longueur (règle SYNC).

Un buffer (règle BUFFER) est bien typé si le type de son entrée est sous-type du type de sa sortie. Nous lions la notion de sous-typage à la notion d'adaptabilité sur les horloges en remarquant qu'un type st est toujours de la forme α `on` stc_1 `on` $\dots$ `on` stc_n. On appelle α sa racine et $stc_1, \dots stc_n$ ses conditions. Les conditions de Core Lucy-n sont des mots ultimement périodiques connues. Leur composition peut donc être calculée. On aboutit à la définition suivante : deux types st et st' sont sous-types (st **<:** st') s'ils ont la même racine et que la composée des conditions de st est adaptable à la composée des conditions de st'. Les définitions ci-dessous expriment formellement cette relation à l'aide de la fonction $Factor(st)$ qui scinde un type d'horloges st en deux : sa variable de type d'une part, la composée de ses conditions d'autre part.

$$
\begin{array}{lcl}
Factor(\alpha) & \triangleq & (\alpha, (1)) \\
Factor(st \texttt{ on } stc) & \triangleq & (\alpha, w \ on \ w') \text{ avec } \left\{ \begin{array}{lcl} (\alpha, w) & = & Factor(st) \\ w' & = & unpack^{\#}([\![stc]\!]^S) \end{array} \right.
\end{array}
$$

$$
st \texttt{ <: } st' \quad \triangleq \quad \alpha = \alpha' \wedge w <: w' \text{ avec } \left\{ \begin{array}{lcl} (\alpha, w) & = & Factor(st) \\ (\alpha', w') & = & Factor(st') \end{array} \right.
$$

La propriété 3 justifie la décidabilité de cette relation. Par exemple, si $st \triangleq \alpha$ `on` (2) `on` $(1\ 0)$ et $st' \triangleq \alpha$ `on` $(0\ 2)$, on a st **<:** st' car $Factor(st) = (\alpha, (1))$, $Factor(st') = (\alpha, (0\ 2))$ et $(1) <: (0\ 2)$.

Le jugement de typage $\Gamma \vdash def :: \Gamma'$ indique que la définition def typée dans l'environnement Γ renvoie un environnement étendu Γ'. Les règles de typage des définitions sont :

$$
\text{NODE} \quad \frac{\Gamma + [x \mapsto st] \vdash e :: st' \qquad FV(\Gamma) = \emptyset}{\Gamma \vdash \texttt{let node } f^{\alpha}\ x \texttt{ = } e :: \Gamma + [f \mapsto Gen(st \to st', \alpha)]}
$$

$$
\text{DEFS} \quad \frac{\Gamma \vdash def_1 :: \Gamma' \qquad \Gamma' \vdash def_2 :: \Gamma''}{\Gamma \vdash def_1; def_2 :: \Gamma''}
$$

Les applications et définitions de nœud fonctionnent comme dans un système à la ML. Les fonctions Gen et $Inst$ permettent de passer d'un type flèche à un schéma de type (règle NODE) et vice-versa (règle APP). Ces fonctions sont définies ci-dessous. La gestion des variables libres et liées étant sans surprises ici, on ne définit pas formellement l'ensemble $FV(ty)$ des variables libres d'un type ty ni l'opération de substitution d'un type ty à une variable de type α dans un type ty' ($ty[ty'/\alpha]$).

$$
\begin{array}{lcl}
Gen(st \to st', \alpha) & \triangleq & \forall \alpha.\ st \ \to \ st' \text{ si } \{\alpha\} = FV(st) \cup FV(st') \\
Inst(\forall \alpha.\ st \ \to \ st',\ st'') & \triangleq & st[st''/\alpha], st'[st''/\alpha]
\end{array}
$$

Notons une particularité : dans un lambda-calcul polymorphe, on doit prendre garde à ne pas généraliser de variable de types libre dans l'environnement de typage. Ici, comme l'environnement ne contient que des types clos après la généralisation, toute variable peut toujours être généralisée.

4.2. Sûreté du typage

Le système de types présenté assure intuitivement la cohérence de la gestion des rafales vis-à-vis de la sémantique de Kahn originale. Plus précisément, pour un programme bien typé, les sémantiques de Kahn et synchrones "coïncident". Nous allons nous concentrer sur les seules règles spécifiques du système de types, c'est à dire WHEN, MERGE, SYNC et BUFFER. Les autres sont essentiellement identiques à celles de ML. Prouver la correction du calcul d'horloge exige d'étudier le comportement des fonctions correspondantes lorsqu'on les applique à des flots finis.

Propriété des combinateurs

Propriété 4. *Appliquée à deux flots finis de même longueur, la fonction when$^\#$ produit un flot fini dont la taille correspond au nombre de booléens à vrai dans sont flot droit :*

$$\forall xs \in S^*, cs \in \mathbb{B}^*, |xs| = |cs| \Rightarrow |when^\#(xs, cs)| = |cs|_{\mathbf{1}}$$

De plus, la fonction when$^\#$ commute avec la concaténation sous les mêmes conditions :

$$\begin{array}{l}\forall xs, xs' \in S^*, cs, cs' \in \mathbb{B}^*, |xs| = |cs| \wedge |xs'| = |cs'| \Rightarrow \\ \quad when^\#(xs.xs', cs.cs') = when^\#(xs, cs).when^\#(xs', cs')\end{array}$$

Propriété 5. *Appliquée à trois flots finis dont le second (resp. troisième) est d'une longueur égale au nombre de booléens à vrai (resp. faux) dans le premier, la fonction merge$^\#$ produit un flot fini de longueur égale à celle de son premier argument :*

$$\forall cs \in \mathbb{B}^*, xs, ys \in S^*, |xs| = |cs|_{\mathbf{1}} \wedge |ys| = |cs|_{\mathbf{0}} \Rightarrow |merge^\#(cs, xs, ys)| = |cs|$$

De plus, elle commute avec la concaténation sous les mêmes conditions :

$$\begin{array}{l}\forall xs, xs', ys, ys' \in S^*, cs, cs' \in \mathbb{B}^*, |xs| = |cs|_{\mathbf{1}} \wedge |ys| = |cs|_{\mathbf{0}} \wedge |xs'| = |cs'|_{\mathbf{1}} \wedge |ys'| = |cs'|_{\mathbf{0}} \Rightarrow \\ \quad merge^\#(cs.cs', xs.xs', ys.ys') = merge^\#(cs, xs, ys).merge^\#(cs', xs', ys')\end{array}$$

Propriété 6. *Appliquée à deux flots finis de même longueur, la fonction sync$^\#$ produit un flot fini de longueur identique :*

$$\forall xs, ys \in S^*, |xs| = |ys| \Rightarrow |sync^\#(xs, ys)| = |xs|$$

De plus, elle commute avec la concaténation sous les mêmes conditions :

$$\begin{array}{l}\forall xs, xs' \in S^*, ys, ys' \in \mathbb{B}^*, |xs| = |ys| \wedge |xs'| = |ys'| \Rightarrow \\ \quad sync^\#(xs.xs', ys.ys') = sync^\#(xs, ys).sync^\#(xs', ys')\end{array}$$

Propriété 7. *Si l'horloge de l'entrée d'un buffer est adaptable à son horloge de production, cette dernière est identique à l'horloge de sa sortie.*

$$\forall xs \in S^*, w \in \mathbb{N}^\omega, clock^\#(xs) <: w \Rightarrow clock^\#(buffer^\#(w, xs)) = w$$

De plus, la sortie d'un buffer est toujours un préfixe de son entrée :

$$\forall xs \in S^*, w \in \mathbb{N}^\omega, unpack^\#(buffer^\#(w, xs)) \sqsubseteq unpack^\#(xs)$$

Propriétés du calcul d'horloge Un système de types ordinaire approxime des ensembles de valeurs (objets sémantiques) par leurs types (objets syntaxiques). Le calcul d'horloge présenté approxime les horloges par des types ; il est naturel de se demander si cette approximation est correcte. Une première idée est de vérifier que l'horloge de la sémantique de toute expression bien typée est égale à la sémantique de son type d'horloges : informellement, si e est de type st dans Γ alors pour tout environnement σ "raisonnable" (respectant Γ), la relation $clock^\#([\![e]\!]^S(\sigma)) = [\![st]\!]^S(\sigma)$ est vérifiée. Cependant, la présence de points fixes dans le langage affaiblit cette égalité. Par exemple, soit l'expression close $e =$ `x where rec x = x` et un environnement σ tel que $\sigma_{st}(\alpha) = (1)$. Selon le calcul d'horloge, e admet n'importe quel type d'horloges, et en particulier $\vdash e :: \alpha$, pourtant $clock^\#([\![e]\!]^S(\sigma)) = clock^\#(fix_{V_S}(\lambda x.x)) = clock^\#(\varepsilon) = (0) \neq [\![\alpha]\!]^S(\sigma) = \sigma_{st}(\alpha) = (1)$.

Le type d'horloges d'une expression est donc une surapproximation. Étant données deux horloges w et w', on dit que w approxime w' (noté $w \leq_{ck} w'$) si w et w' sont égales pour toujours ou bien jusqu'à ce que w contienne une infinité de zéros :

$$\begin{array}{lll}(0) & \leq_{ck} & w \\ n.w & \leq_{ck} & n.w' \text{ pour tout } n, w, w' \text{ tels que } w \leq_{ck} w'\end{array}$$

On peut maintenant définir la relation $\sigma \models_\tau x$ qui signifie informellement que dans l'environnement synchrone σ, l'horloge de l'objet (valeur synchrone ou fonction) x est approximée par le type composé ou schéma de type τ. Cette relation est définie inductivement sur les types composés :

$$\begin{array}{lcl}
\sigma \models_{st} r & \triangleq & clock^\#(r) \leq_{ck} [\![st]\!]^S(\sigma) \\
\sigma \models_{ty \times ty} (r, r') & \triangleq & \sigma \models_{ty} r \wedge \sigma \models_{ty'} r' \\
\sigma \models_{\forall \alpha.\ ty \to ty'} f & \triangleq & \forall w, r.\ (\sigma' \models_{ty} r) \Rightarrow (\sigma' \models_{ty'} f\ w\ r) \\
 & & \text{avec } \sigma' = \sigma + [\alpha \mapsto w]
\end{array}$$

Un environnement de typage Γ et un environnement synchrone σ sont dits *cohérents* (noté $\Gamma \perp \sigma$) si les horloges des valeurs et fonctions contenues dans σ correspondent à la sémantique des types et schémas de types des valeurs et nœuds dans Γ. En d'autres termes :

$$\Gamma \perp \sigma \quad \triangleq \quad \forall (x, ty) \in \Gamma,\ \sigma \models_{ty} \sigma(x) \ \wedge\ \forall (f, ty_{sch}) \in \Gamma,\ \sigma \models_{ty_{sch}} \sigma(f)$$

Ces définitions permettent de définir la première propriété du système de types.

Théorème 1 (Cohérence)**.** *L'horloge d'une expression bien typée est décrite par son type.*

$$\forall e, ty, \Gamma, \sigma,\ (\Gamma \perp \sigma \wedge \Gamma \vdash e :: ty) \Rightarrow \sigma \models_{ty} [\![e]\!]^S(\sigma)$$

Enfin, on définit formellement la notion d'équivalence entre les deux sémantiques. Intuitivement, un flot de rafales $s \in (S^*)^\infty$ et un flot de scalaires $s \in S^\infty$ sont équivalents si en aplatissant les rafales du premier on obtient un préfixe du second. La relation $\lhd$ étend cette relation aux types composés et schémas de types :

$$\begin{array}{lcl}
r \lhd_{st} s & \triangleq & unpack^\#(r) \sqsubseteq s \\
(r, s) \lhd_{(ty,ty')} (r', s') & \triangleq & r \lhd_{ty} s \wedge r' \lhd_{ty'} s' \\
f \lhd_{\forall \alpha_1, \ldots, \alpha_n.\ ty \to ty'} g & \triangleq & \forall w_1, \ldots, w_n, x_S, x_K,\ (x_S \lhd_{ty} x_K \Rightarrow f\ w_1\ \ldots\ w_n\ x_S \lhd_{ty'} g\ x_K)
\end{array}$$

Nous voulons maintenant énoncer formellement la correction du calcul d'horloge. La relation $\lhd$ doit pour cela être étendue aux environnements. Un environnements de Kahn et un environnement synchrone définissant les mêmes variables sont cohérents (pour $\lhd$) lorsqu'ils s'entendent sur les valeurs des variables par rapport à un environnement de typage Γ donné :

$$\sigma_S \approx_\Gamma \sigma_K \quad \triangleq \quad \forall (x, ty) \in \Gamma,\ \sigma_S(x) \lhd_{ty} \sigma_K(x)$$

Nous pouvons maintenant formaliser la définition de la correction du typage donnée plus haut.

Théorème 2 (Correction)**.** *La sémantique synchrone d'une expression bien typée approxime sa sémantique de Kahn :*

$$\forall e, ty, \Gamma, \sigma_S, \sigma_K,\ (\Gamma \vdash e :: ty \wedge \sigma_S \approx_\Gamma \sigma_K) \Rightarrow [\![e]\!]^S(\sigma_S) \lhd_{ty} [\![e]\!]^K(\sigma_K)$$

Démonstration. La preuve se fait par induction sur les dérivations du calcul d'horloge. On esquisse quelques cas intéressants :

- Les sémantiques synchrones des opérateurs `when`, `merge` et `sync` ont la même forme. Par hypothèse d'induction, on doit prouver, pour $f \in \{when^\#, merge^\#, sync^\#\}$,

  $$unpack^\#(map^\#_n(f, pack^\#(w_1, s_1), \ldots, pack^\#(w_n, s_n))) \sqsubseteq f\ s_1\ \ldots\ s_n$$

  ce que le théorème 1 et les deuxièmes parties des propriétés 4, 5 et 6 justifient.
- La deuxième partie de la propriété 7 permet prouver la correction des buffers.
- La généralisation des variables de type est correcte grâce à la propriété 2 : pour prouver α `on` p `<:` α `on` p', vérifier $p <: p'$ suffit.

□

5. Perspectives d'implantation

Nous sommes en train de réaliser un prototype de compilateur pour un langage proche de Core Lucy-n. On discute ici de quelques aspects non discutés dans le reste de l'article mais nécessaires à la conception d'un compilateur complet.

Inférence d'horloge Le calcul d'horloge présenté en section 4 vérifie la cohérence du programme. Comme indiqué en section 2, notre prototype est capable d'inférer des horloges entières. Il utilise actuellement un algorithme développé pour les horloges binaires dans l'article [8], qui s'étend naturellement au cadre entier. Nous étudions actuellement des raffinements plus efficaces inspirés des techniques utilisées sur les *Synchronous Data-Flow* [7]. Par ailleurs, le prototype accepte des conditions arbitraires à la Lucid Synchrone et ne se restreint donc pas aux horloges ultimement périodiques.

Causalité et génération de code Les compilateurs pour langages synchrones emploient une analyse dite de *causalité* pour rejeter les programmes comportant des interblocages. Dans certains cas, cette analyse assure également que les équations du source peuvent être ordonnées statiquement sans introduire d'interblocage. Cela permet de générer du code impératif séquentiel.

Dans les langages synchrones fonctionnels, la causalité est une analyse de *productivité* : on cherche à s'assurer que chaque expression, définitions récursives comprises, produit des flots infinis. En pratique, on assure que la définition d'une variable ne dépend d'elle-même qu'à travers un délai. L'opérateur de délai est spécifique à chaque langage.

Dans notre cadre, cette notion est délicate. On peut, comme en Lucy-n [10], décider qu'un délai est un buffer n'ayant jamais besoin de son entrée à l'instant courant pour produire sa sortie. On appelle *adaptabilité stricte* ($\ll:$) la relation suivante entre les horloges d'entrée et sortie d'un buffer :

$$w \ll: w' \triangleq w <: w' \wedge \forall i \in \mathbb{N}, \mathcal{O}_w(i) \geq \mathcal{O}_{w'}(i+1)$$

Néanmoins, cette notion ne se comporte *a priori* pas bien vis-à-vis de la généralisation des variables de type. En effet, on souhaite avoir une propriété semblable à la propriété 2 pour l'adaptabilité stricte : si $w \ll: w'$ alors w'' `on` $w \ll: w''$ `on` w'. Malheureusement, cette propriété est fausse. Par exemple, on a $(1) \ll: 0(1)$ mais pas (2) `on` $(1) \ll: (2)$ `on` $0(1)$ car $(2) \not\ll: 1(2)$.

Pour vérifier nœud par nœud la causalité des programmes, plusieurs solutions sont envisageables. La première est de complexifier le système de types pour introduire de la quantification bornée. Tous les jugements α `on` *stc* $\ll:$ α `on` *stc'* vérifiés localement ajoutent au type du nœud une contrainte sur α. Cette contrainte doit être telle que tout type la respectant ne falsifie pas le jugement initial. Cette contrainte dépendante de *stc* et *stc'* reste à définir.

Une autre approche explorée par le prototype est d'exploiter un artefact du processus de génération de code. Pour présenter celle-ci, nous expliquons grossièrement le processus de génération de code.

Comme illustré en section 2, les compilateurs pour Lucid Synchrone et SCADE 6 génèrent pour chaque nœud une unique fonction de transition dans un langage en appel par valeur. Ce choix est un compromis : il permet une génération de code et analyse de causalité plus simples, mais rejette des programmes valides (comme `x where rec x = f x`, avec `f` un nœud dont la sortie ne dépend pas instantanément de l'entrée).

Nous avons choisi de suivre cette approche, et de fixer la taille des rafales en entrée et sortie d'une transition. Chaque rafale est représentée par un tableau. Par exemple, la fonction de transition générée depuis un nœud `f` de type $\forall\alpha.\ \alpha$ `on` $(3) \rightarrow \alpha$ `on` (2) reçoit et produit des tableaux de tailles respectives trois et deux. Cela implique qu'à chaque application de `f`, le compilateur génère du code de contrôle et convertisse les tableaux en entrée et sortie.

Par exemple, imaginons le code généré par un appel `y = f x`, avec `x` de type α' `on` (6) et `y` de type α' `on` (4). Cela correspond à instancier α avec α' `on` (2) dans le type de `f`. Cet appel est traduit

en une boucle de deux itérations autour d'un appel à la fonction de transition `f_step` de `f`, ainsi qu'en du code convertissant `x` et `y` en des tableaux de tailles attendues par `f_step`. Ici, il faut découper le tableau de taille six correspondant à `x` en deux tableaux de taille trois, et concaténer les deux sorties de taille deux en un tableau de taille quatre correspondant à `y`.

Cette méthode de génération de code esquive le problème de causalité évoqué précédemment. Intuitivement, le fait que w'' `on` $w \ll: w''$ `on` w' ne dépende pas uniquement de w et w' provient du fait que w'' peut être "trop grand" ; générer le code de contrôle et conversion autour de la fonction de transition revient à ne jamais instancier l'horloge de base du buffer qu'avec (1) : on n'y produit ni ne consomme jamais de rafale d'une taille différente de celles induites par w ou w'. Notre compilateur se contente donc de l'adaptabilité stricte à la Lucy-n. Nous n'avons pas encore d'explication sémantique convaincante de ce phénomène.

Conclusion Nous avons proposé un langage qui étend les modèles synchrones et n-synchrones et brièvement décrit un embryon de compilateur. Formaliser la génération de code devrait aider à clarifier ses subtilités.

Remerciements Nous remercions Guillaume Baudart, Albert Cohen, Nhat Minh Lê, Marc Pouzet et Gabriel Scherer pour leur relecture.

Références

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[2] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the conference on Languages, compilers, and tools for embedded systems*, 2008.

[3] P. Caspi and M. Pouzet. Réseaux de Kahn Synchrones. In *Journées Francophones des Langages Applicatifs*, 1996.

[4] A. Cohen, P. Dumont, M. Duranton, and M. Pouzet. Horloges entières. Draft, 2007.

[5] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. *N*-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *ACM International Conference on Principles of Programming Languages*, 2006.

[6] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475. North Holland, Amsterdam, 1974.

[7] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, Jan. 1987.

[8] L. Mandel and F. Plateau. Typage des horloges périodiques en Lucy-n. In *Vingt deuxièmes Journées Francophones des Langages Applicatifs*, 2011.

[9] L. Mandel, F. Plateau, and M. Pouzet. Lucy-n : une extension n-synchrone de Lustre. In *Vingt et unièmes Journées Francophones des Langages Applicatifs*, 2010.

[10] F. Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée.* PhD thesis, Université Paris-Sud 11, 2010.

Pretty-big-step-semantics-based Certified Abstract Interpretation

Martin Bodin[1,2], Thomas Jensen[2], & Alan Schmitt[2]

1: ENS Lyon; 2: Inria; `firstname.lastname@inria.fr`

Résumé

We present a technique for deriving semantic program analyses from a natural semantics specification of the programming language. The technique is based on the pretty-big-step semantics approach applied to a language with simple objects called O'While. We specify a series of instrumentations of the semantics that makes explicit the flows of values in a program. This leads to a semantics-based dependency analysis, at the core, *e.g.*, of tainting analysis in software security. The formalization is currently being done with the Coq proof assistant.[1]

1. Introduction

David Schmidt gave an invited talk at the 1995 Static Analysis Symposium [11] in which he argued for using natural semantics as a foundation for designing semantic program analyses within the abstract interpretation framework. With natural (or "big-step" or "evaluation") semantics, we can indeed hope to benefit from the compositional nature of a denotational-style semantics while at the same time being able to capture intentional properties that are best expressed using an operational semantics. Schmidt showed how a control flow analysis of a core higher-order functional language can be expressed elegantly in his framework. Subsequent work by Gouranton and Le Métayer showed how this approach could be used to provide a natural semantics-based foundation for program slicing [13].

In this paper, we will pursue the research agenda set out by Schmidt and investigate further the systematic design of semantics-based program analyses based on big-step semantics. Two important issues here will be those of scalability and mechanization. The approach worked nicely for a language whose semantics could be defined in 8 inference rules. How will it react when applied to full-blown languages where the semantic definition comprises hundreds of rules? Strongly linked to this question is that of how the framework can be mechanized and put to work on larger languages using automated tool support. In the present work, we investigate how the Coq proof assistant can serve as a tool for manipulating the semantic definitions and certifying the correctness of the derived static analyses.

Certified static analysis is concerned with developing static analyzers inside proof assistants with the aim of producing a static analyzer and a machine-verifiable proof of its semantic correctness. One long-term goal of the work reported here is to be able to provide a mechanically verified static analysis for the full JavaScript language based on the Coq formalization developed in the JSCert project [1]. JavaScript, with its rich but sometimes quirky semantics, is indeed a good *raison d'être* for studying certified static analysis, in order to ensure that all of the cases in the semantics are catered for.

In our development, we shall take advantage of some recent developments in the theory of operational semantics. In particular, we will be using a particular format of natural semantics call

[1]This work has been presented at the Festschrift for David Schmidt in September 2013. This work has been partially supported by the French National Research Agency (ANR), project Typex ANR-11-BS02-007, and by the Laboratoire d'excellence CominLabs ANR-10-LABX-07-01.

$s ::=$
| `skip`
| $s_1 ; s_2$
| `if` e `then` s_1 `else` s_2
| `while` e `do` s
| `x =` e
| e_1`.f =` e_2
| `delete` e`.f`

$e ::=$
| c
| `x`
| e_1 `op` e_2
| `{}`
| e`.f`

$v ::=$
| c
| l

$r ::=$
| S
| S, v
| $S,$ `err`

$s_e ::=$
| s
| $r \;;_1 s$
| `if1`(r, s_1, s_2)
| `while1`(r, e, s)
| `while2`(r, e, s)
| `x` $=_1 r$
| r`.f` $=_1 e$
| l`.f` $=_2 r$
| `delete1` r`.f`

$e_e ::=$
| e
| r `op`$_1$ e
| v `op`$_2$ r
| r `.`$_1$`f`

Figure 1: O'While Syntax, Values, Results, and Extended Syntax

"pretty-big-step" semantics [3] which is a streamlined form of operational semantics retaining the format of natural semantics while being closer to small-step operational semantics.

Even though it is our ultimate goal, JavaScript is far too big to begin with as a goal for analysis: its pretty-big-step semantics contains more than half a thousand rules! We will thus start by studying a much simpler language, called O'While, which is basically a While language with simple objects in the form of extensible records. This language is quite far from JavaScript, but is big enough to catch some issues of the analyses of JavaScript objects. We present the language and its pretty-big-step semantics in Section 2. To test the applicability of the approach to defining static analyses, we have chosen to formalize a data flow dependency analysis as used, *e.g.*, in tainting [12] or "direct information-flow" analyses of JavaScript [15, 5]. The property we ensure is defined in Section 3 and the analysis itself is defined in Sections 4. As stated above, the scalability of the approach relies on the mechanization that will enable the developer of the analyses to prove the correctness of analyses with respect to the semantics, and to extract an executable analyzer. We show how the Coq proof assistant is currently being used to formally achieve these objectives as we go along.

2. O'While and its Pretty Big Step Semantics

As big-step semantics, pretty-big-step semantics directly relates terms to their results. However, pretty-big-step semantics avoids the duplication associated with big-step semantics when features such as exceptions and divergence are added. Since duplication in the definitions often leads to duplication in the formalization and in the proofs, an approach based on a pretty-big-step semantics allows to deal with programming languages with many complex constructs. (We refer the reader to Charguéraud's work on pretty-big-step semantics [3] for detailed information about this duplication.) Even though the language considered here is not complex, we have been using pretty-big-step semantics exclusively for our JavaScript developments, thus we will pursue this approach in the present study.

The syntax of O'While is presented in Figure 1. Two new constructions have been added to the syntax of expressions for the usual While language: `{}` creates a new object, and e`.f` accesses a field of an object. Regarding statements, we allow the addition or the modification of a field to an object using e_1`.f =` e_2, and the deletion of the field of an object using `delete` e`.f`. In the following we write t for terms, *i.e.*, both expressions and statements.

Objects are passed by reference. Values v are either locations l or primitive values c. In this work,

we only consider boolean primitive values. The *state* of a program contains both an *environment* E, which is a mapping from variables to values, and a *heap* H, which is a mapping from locations to objects, that are themselves mappings from fields to values. In the following, we write S for E, H when there is no need to access the environment nor the heap. Results r are either a state S, a pair of a state and value S, v, or a pair of an error and a state $S, \texttt{err}$.

Figure 1 also introduces *extended statements* and *extended expressions* that are used in O'While's pretty-big-step semantics, presented in Figure 2. Extended terms t_e comprise extended statements and expressions. Reduction rules have the form $S, t_e \to r$. The result r can be an error $S', \texttt{err}$. Otherwise, if t_e is an extended statement, then r is a state S', and if t_e is an extended expression, then r is a pair of a state S' and returned value v. We write $\texttt{st}(r)$ for the state S in a result r.

Most rules are the usual While ones, with the exception that they are given in pretty-big-step style. We now detail the new rules for expressions and statements. Rule Obj associates an empty object to a fresh location in the heap. Rule Fld for the expression $e.\texttt{f}$ first evaluates e to some result r, then calls the rule for the extended expression $r\,._1\texttt{f}$. The rule for this extended expression is only defined if r is of the form E', H', l where l is a location in H' that points to an object o containing a field $\texttt{f}$. The rules for field assignment and field deletion are similar: we first evaluate the expression that defines the object to be modified, and in the case it actually is a location, we modify this object using an extended statement.

Finally, our semantics is parameterized by a partial function $\texttt{abort}(\cdot)$ from extended terms to results, that indicates when an error is to be raised or propagated. More precisely, the function $\texttt{abort}(t_e)$ is defined at least if t_e is an extended term containing a subterm equal to $S, \texttt{err}$ for some S. In this case $\texttt{abort}(t_e) = S, \texttt{err}$. We can then extend this function to define erroneous cases. For instance, we could say that $\texttt{abort}((E, H, v).\texttt{f} =_1 e) = E, H, \texttt{err}$ if v is not a location, or if $v = l$ but l is not in the domain of H, or if $\texttt{f}$ is not in the domain of $H[l]$. This function is used in the Abort rule, that defines when an error is raised or propagated. This illustrates the benefit of a pretty-big-step semantics: a single rule covers every possible error propagation case.

The derivation in Figure 14 in Appendix B is an example of a derivation of the semantics.

3. Annotated Semantics

3.1. Execution traces

We want to track how data created at one point flows into locations (variables or object fields) at later points in the execution of the program. To this end, we need a mechanism for talking about "points of time" in a program execution. This information is implicit in the semantic derivation tree corresponding to the execution. To make it explicit, we instrument the semantics to produce a (linear) trace of the inference rules used in the derivation, and use it to refer to particular points in the execution. As every other instrumentation in this paper, it adds no information to the derivation but allows global information to be discussed locally.

More precisely, we add partial traces, $\tau \in \mathit{Trace}$, to both sides of the reduction rules. These traces are lists of names used in the derivation. The two crucial properties from traces is that they uniquely identify a point in the derivation (i.e., a rule in the tree and a side of this rule), and that one may derive from the trace the syntactic program point that is being executed at that point.

Since traces uniquely identify places in a derivation, we use them from now on to refer to states or further instrumentation in the derivation. More precisely, if τ is a trace in a given derivation, we write E_τ and H_τ for the environment and heap at that point.

$$\frac{}{S, \texttt{skip} \to S}\ \textsc{Skip} \qquad \frac{S, s_1 \to r \qquad S, r\ ;_1\ s_2 \to r'}{S, s_1; s_2 \to r'}\ \textsc{Seq} \qquad \frac{S', s \to r}{S, S'\ ;_1\ s \to r}\ \textsc{Seq1}$$

$$\frac{S, e \to r \qquad S, \texttt{if1}(r, s_1, s_2) \to r'}{S, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to r'}\ \textsc{If} \qquad \frac{S', s_1 \to r}{S, \texttt{if1}((S', \texttt{true}), s_1, s_2) \to r}\ \textsc{IfTrue}$$

$$\frac{S', s_2 \to r}{S, \texttt{if1}((S', \texttt{false}), s_1, s_2) \to r}\ \textsc{IfFalse} \qquad \frac{S, e \to r \qquad S, \texttt{while1}(r, e, s) \to r'}{S, \texttt{while}\ e\ \texttt{do}\ s \to r'}\ \textsc{While}$$

$$\frac{S', s \to r \qquad S', \texttt{while2}(r, e, s) \to r'}{S, \texttt{while1}((S', \texttt{true}), e, s) \to r'}\ \textsc{WhileTrue1} \qquad \frac{S', \texttt{while}\ e\ \texttt{do}\ s \to r}{S, \texttt{while2}(S', e, s) \to r}\ \textsc{WhileTrue2}$$

$$\frac{}{S, \texttt{while1}((S', \texttt{false}), e, s) \to S'}\ \textsc{WhileFalse} \qquad \frac{E, H, e \to r \qquad E, H, \texttt{x} =_1 r \to r'}{E, H, \texttt{x = } e \to r'}\ \textsc{Asg}$$

$$\frac{E' = E[\texttt{x} \mapsto v]}{S, \texttt{x} =_1 (E, H, v) \to E', H}\ \textsc{Asg1} \qquad \frac{S, e_1 \to r \qquad S, r.\texttt{f} =_1 e_2 \to r'}{S, e_1.\texttt{f = } e_2 \to r'}\ \textsc{FldAsg}$$

$$\frac{S', e \to r \qquad S', l.\texttt{f} =_2 r \to r'}{S, (S', l).\texttt{f} =_1 e \to r'}\ \textsc{FldAsg1} \qquad \frac{H[l] = o \qquad o' = o\,[\texttt{f} \mapsto v] \qquad H' = H\,[l \mapsto o']}{S, l.\texttt{f} =_2 (E, H, v) \to E, H'}\ \textsc{FldAsg2}$$

$$\frac{S, e \to r \qquad S, \texttt{delete1}\ r.\texttt{f} \to r'}{S, \texttt{delete}\ e.\texttt{f} \to r'}\ \textsc{Del}$$

$$\frac{H[l] = o \qquad o[\texttt{f}] \neq \bot \qquad o' = o\,[\texttt{f} \mapsto \bot] \qquad H' = H\,[l \mapsto o']}{S, \texttt{delete1}\ (E, H, l).\texttt{f} \to E, H'}\ \textsc{Del1} \qquad \frac{}{S, c \to S, c}\ \textsc{Cst}$$

$$\frac{E[\texttt{x}] = v}{E, H, \texttt{x} \to E, H, v}\ \textsc{Var} \qquad \frac{S, e_1 \to r \qquad S, r\ \texttt{op}_1\ e_2 \to r'}{S, e_1\ \texttt{op}\ e_2 \to r'}\ \textsc{Bin} \qquad \frac{S', e_2 \to r \qquad S', v_1\ \texttt{op}_2\ r \to r'}{S, (S', v_1)\ \texttt{op}_1\ e_2 \to r'}\ \textsc{Bin1}$$

$$\frac{v = v_1\ \texttt{op}\ v_2}{S, v_1\ \texttt{op}_2\ (S, v_2) \to S, v}\ \textsc{Bin2} \qquad \frac{H[l] = \bot \qquad H' = H[l \mapsto \{\}]}{E, H, \{\} \to E, H', l}\ \textsc{Obj} \qquad \frac{S, e \to r \qquad S, r\ ._1\texttt{f} \to r'}{S, e.\texttt{f} \to r'}\ \textsc{Fld}$$

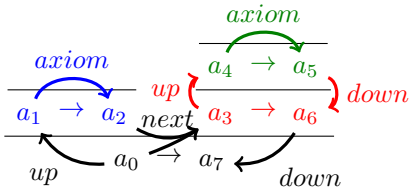
$$\frac{H'[l] = o \qquad o[\texttt{f}] = v}{E, H, (E', H', l)\ ._1\texttt{f} \to E', H', v}\ \textsc{Fld1} \qquad \frac{\texttt{abort}(t_e) = r}{S, t_e \to r}\ \textsc{Abort}$$

Figure 2: O'While's Semantics

3.2. A General Scheme to Define Annotations

In principle, the annotation process takes as argument a full derivation tree and returns an annotated tree. However, every annotation process we define in the following, as well as the one deriving traces, can be described by an iterative process that takes as arguments previous annotations and the parameters of the rule applied, and returns an annotated rule.

More precisely, our iterative process is based on steps of four kinds: *axiom* steps (for axioms), that transform the annotations on the left of axiom rules into annotations on the right of the rule, *up* steps (for rules with inductive premises), that propagate an annotation on the left of a rule to its first premise, *down* steps (for rules with inductive premises), that propagate an annotation on the right of the last premise to the right of the rule, and *next* steps (for rules with two inductive premises), that propagate the annotations from the left of the current rule and from the right of the first premise into the left of the second premise. As we are using a pretty-big-step semantics, there are at most two inductive premises above each rule, thus these steps are sufficient.



This generic approach allows to compose complex annotations, building upon previously defined ones. This general scheme is summed up on the right, where each a_i represents an annotation. The colors show which steps are associated to which rules: a_0 is the initial annotation. It is changed to a_1 and control is passed to the left axiom rule (black *up*). The blue *axiom* step creates a_2, and control returns to the bottom rule, where the black *next* step combines a_2 and a_0 to pass it to the red rule. Annotations are propagated in the right premise, and ultimately control comes back to the black rule which pulls the a_6 annotation from the red rule and creates its a_7 annotation. Note that the types of the annotations on the left and the right of the rules do not have to be the same, as long as every left-hand side annotation has the same type, and the same for right-hand side annotations.

As an example, we define the *axiom*, *up*, *down*, and *next* steps corresponding to the addition of partial traces for rules Var and Asg (see Figure 3b and 4b). Rule Var illustrates the *axiom* step, that adds a Var token at the end of the trace. Rule Asg illustrates the other steps: adding a AsgE as *up* step, adding a $\overline{\text{AsgE}}$ as *next* step, and adding Asg as *down* step. We fully describe in Appendix A.2 how the traces are added following this approach.

3.3. Dependency Relation

We are interested in deriving the dependency analysis underlying tainting analyses for checking that secret values do not flow into other values that are rendered public. To this end, we consider *direct flows* from *sources* to *stores*. We need a mechanism for describing when data was created and when a flow happened, so we annotate locations in the heap with the time when they were allocated. By "time" we here mean the point of time in an execution, represented by a trace τ of the derivation. We write $ALoc = Loc \times Trace$ for the set of *annotated locations*. Similarly, we annotate variables and fields with the point in time that they were last assigned to. When describing a flow, we talk about *sources* and *stores*. Sources are of three kinds: an annotated location, a variable annotated with its last modification time, or a pair of annotated location and field further annotated with their last modification time. Stores are either a variable or a pair of an annotated location and a field, further annotated with their last modification time. Formally, we define the following dependency relation

$$\leftsquigarrow \in Dep = \mathcal{P}\left(Source \times Store\right)$$

where $Store = (Var \times Trace) + (ALoc \times Field \times Trace)$ and $Source = ALoc + Store$.

For instance, we write $\mathtt{y}^{\tau_1} \leftsquigarrow \mathtt{x}^{\tau_2}$ to indicate that the content that was put in the variable `y` at time τ_1 has been used to compute the value stored in the variable `x` at time τ_2. Similarly, we write

$$\textsc{Var}\ \frac{E[\underline{\mathtt{x}}] = v}{E, H, \underline{\mathtt{x}} \to E, H, v}$$

(a) Basic Rule

$$\frac{E[\underline{\mathtt{x}}] = v \qquad \tau' = \tau +\!\!+ [\textsc{Var}]}{E, H, \tau, \underline{\mathtt{x}} \to E, H, \tau', v}\ \textsc{Var}$$

(b) Adding Partial Traces

$$\textsc{Var}\ \frac{E[\underline{\mathtt{x}}] = v \qquad \tau' = \tau +\!\!+ [\textsc{Var}]}{E, H, \tau, M, \underline{\mathtt{x}} \to E, H, \tau', M, v}$$

(c) Adding Last-Modified Place

$$\frac{E[\underline{\mathtt{x}}] = v \qquad \tau' = \tau +\!\!+ [\textsc{Var}] \qquad M[\mathtt{x}] = \tau_0}{E, H, \tau, M, \underline{\mathtt{x}} \to E, H, \tau', M, \{\mathtt{x}^{\tau_0}\}, v}\ \textsc{Var}$$

(d) Adding Dependencies

Figure 3: Instrumentation Steps for Var

$l^{\tau_2} \leftsquigarrow l'^{\tau_1}.\mathtt{f}^{\tau_3}$ to indicate that the object allocated at location l at time τ_2 flows at time τ_3 into field $\mathtt{f}$ of location l' that was allocated at time τ_1.

3.4. Direct Flows

We now detail how to compose additional annotations to define our direct flow property $\leftsquigarrow$. As flows are a global property of the derivation, we use a series of annotations to propagate local information until we can locally define direct flows.

We first collect in the derivation the traces where locations are created and where variables or object fields are assigned. To this end, we define a new annotation M of type $(Loc + Var + ALoc \times Field) \to Trace$. After this instrumentation step, reductions are of the form $\tau, M_\tau, S_\tau, t \to \tau', M_{\tau'}, r$. Note that the trace in information in $ALoc \times Field$ is redundant in our setting, as locations may not be reused. It is however useful when showing the correspondence with the analyses as the trace information lets us derive the program point at which the location was allocated.

The three rules that modify M are Obj, Asg1, and FldAsg2. We describe them in Figure 5. The other rules simply propagate M. For the purpose of our analysis, we do not consider the deletion of a field as its modification. More precise analyses, in particular ones that also track indirect flows, would need to record such events.

The added instrumentation uses traces to track the moments when locations are created, and when fields and variables are assigned. For field assignment, the rule FldAsg2 relies on the fact that the location of the object assigned has already been created to obtain the annotated location: we have the invariant that if $H[l]$ is defined, then $M[l]$ is defined.

We can now continue our instrumentation by adding *dependencies* $d \in \mathcal{P}(Var)$. The instrumented reduction is now $\tau, M_\tau, d_\tau, S_\tau, t \to \tau', M_{\tau'}, d_{\tau'}, r$. Its rules are described in Figure 6. The rules not given only propagate the dependencies. The intuition behind these rules is that expressions generate potential dependencies that are thrown away when they don't result in direct flow (for instance when computing the condition of a If statement). The important rules are Var, where the result depends on the last time the variable was modified, Obj, which records the dependency on the creation of the object, and Fld1, whose result depends on the last time the field was assigned. The Asg and FldAsg1 rules make sure these dependencies are transmitted to the inductive call to the rule that will proceed with the assignment for the next series of annotations.

Finally, we build upon this last instrumentation to define flows. The final instrumented derivation is of the form: $\tau, M_\tau, d_\tau, \Delta_\tau, S_\tau, s \to \tau', M_{\tau'}, d_{\tau'}, \Delta_{\tau'}, r_{\tau'}$, where $\{\Delta_\tau, \Delta_{\tau'}\} \subseteq Dep$ are sets of flows defining the $\leftsquigarrow$ relation (see Section 3.3). The two important rules are Asg1 and FldAsg2, which modify respectively a variable and a field, and for which the flow needs to be added. All the other

$$\frac{S, e \to r \qquad S, \underline{\texttt{x} =_1 r} \to r'}{S, \underline{\texttt{x = } e} \to r'}\ \textsc{Asg}$$

(a) Basic Rule

$$\tau_1 = \tau_0 +\!\!+ [\textsc{Asg}\,\mathrm{E}] \qquad \tau_3 = \tau_2 +\!\!+ [\overline{\textsc{Asg}\,\mathrm{E}}] \qquad \tau_5 = \tau_4 +\!\!+ [\textsc{Asg}]$$

$$\frac{\tau_1, S, e \to \tau_2, r \qquad \tau_3, S, \underline{\texttt{x} =_1 r} \to \tau_4, r'}{\tau_0, \underline{\texttt{x = } e}, \to \tau_5, r'}\ \textsc{Asg}$$

(b) Adding Partial Traces

$$\frac{\tau_1, M, S, e \to \tau_2, M', r \qquad \tau_3, M', S, \underline{\texttt{x} =_1 r} \to \tau_4, M'', r'}{\tau_0, M, \underline{\texttt{x = } e}, \to \tau_5, M'', r'}\ \textsc{Asg}$$

(c) Adding Last-Modified Place

$$\frac{\tau_1, M, \emptyset, \Delta, S, e \to \tau_2, M', d, \Delta, r \qquad \tau_3, M', d, \Delta, S, \underline{\texttt{x} =_1 r} \to \tau_4, M'', \emptyset, \Delta', r'}{\tau_0, M, \emptyset, \Delta, S, \underline{\texttt{x = } e} \to \tau_5, M'', \emptyset, \Delta', r'}\ \textsc{Asg}$$

(d) Adding Dependencies

Figure 4: Instrumentation Steps for Asg

rules just propagate those new constructions. The two modified rules are given in Figure 7.

3.5. Correctness Properties of the Annotations

The instrumentation of the semantics does not add information to the reduction but only makes existing information explicit. The correctness of the instrumentation can therefore be expressed as a series of consistency properties between the different instrumented semantics.

We first state correctness properties about the instrumentation of the heap. We start by a property concerning the last-modified-place annotations. This property states that the annotation of a location's creation point never changes, and that the value of a field has not changed since the point of modification indicated by the instrumentation component M.

Property 1 *For every instrumented tree, and for every rule in this tree $\tau, M_\tau, E_\tau, H_\tau, t \to \tau', M_{\tau'}, r$ where $\texttt{st}(r) = E_{\tau'}, H_{\tau'}$ and $M_{\tau'}[l^{\tau_0}.\texttt{f}] = \tau_1$; we have $M_{\tau'}[l] = \tau_0$ and $H_{\tau'}[l][\texttt{f}] = H_{\tau_1}[l][\texttt{f}]$.*

The following property links the last-change-place annotation (M) with the dependencies annotation (Δ). Intuitively, it states that if Δ says that the value assigned to x at time τ_1 later flew into a variable a time τ_2 then x has not changed between τ_1 and τ_2.

Property 2 *For every instrumented tree, and for every rule in this tree $s, \tau, M_\tau, d_\tau, \Delta_\tau, S_\tau, t \to \tau', M_{\tau'}, d_{\tau'}, \Delta_{\tau'}, r$ if $\texttt{x}^{\tau_1} \leftrightsquigarrow \texttt{y}^{\tau_2} \in \Delta_{\tau'}$, then at time τ_2, the last write to $\texttt{x}$ was at time τ_1,* i.e., *$M_{\tau_2}[\texttt{x}] = \tau_1$.*

We now state the most important property: if at some point during the execution of a program the field of an object contains another object, then there is a chain of direct flows attesting it in the

$$\frac{H[l] = \bot \qquad H' = H[l \mapsto \underline{\{\}}] \qquad M' = M[l \mapsto \tau']}{\tau, M, E, H, \underline{\{\}} \to \tau', M', E, H', l}\ \textsc{Obj}$$

$$\frac{H[l] = o \qquad o' = o\,[\mathtt{f} \mapsto v] \qquad H' = H'\,[l \mapsto o'] \qquad M' = M[(l, M[l], f) \mapsto \tau']}{\tau, M, S, \underline{l.\mathtt{f} =_2 (E, H, v)} \to \tau', M', E, H'}\ \textsc{FldAsg2}$$

$$\frac{E' = E[\mathtt{x} \mapsto v] \qquad M' = M[\mathtt{x} \mapsto \tau']}{\tau, M, S, \underline{\mathtt{x} =_1 (E, H, v)} \to \tau', M', E', H}\ \textsc{Asg1}$$

Figure 5: Adding Modified and Created Information

annotation. More precisely, we write $l_0 \leadsto^*_\Delta l_n.\mathtt{f}$ if there are stores $s_0 \ldots s_n$ such that: $s_0 = l_0^{\tau_0}$ for some τ_0, $s_n = l_n^{\tau_n}.\mathtt{f}^{\tau'_n}$ for some τ_n and τ'_n, and for every i in $[1..n]$ we have either $s_i = l_i^{\tau_i}.\mathtt{f}_i{}^{\tau'_i}$ for some l_i, $\mathtt{f}_i$, τ_i, and τ'_i or $s_i = \mathtt{x}_i{}^{\tau'_i}$ for some $\mathtt{x}_i$ and τ'_i; and for every i, $s_i \leadsto s_{i+1} \in \Delta$.

Property 3 *For every instrumented tree, and for every rule in this tree $\tau, M_\tau, d_\tau, \Delta_\tau, E_\tau, H_\tau, t \to \tau', M_{\tau'}, d_{\tau'}, \Delta_{\tau'}, r$ where $\mathtt{st}(r) = E_{\tau'}, H_{\tau'}$, we have: for every locations l, l' and field $\mathtt{f}$ such that $H_\tau[l'][\mathtt{f}] = l$, then $l \leadsto^*_{\Delta_\tau} l'.\mathtt{f}$; for every locations l, l' and field $\mathtt{f}$ such that $H_{\tau'}[l'][\mathtt{f}] = l$, then $l \leadsto^*_{\Delta_{\tau'}} l'.\mathtt{f}$.*

3.6. Annotated Semantics in Coq

In the Coq development, we distinguish expressions from statements, and we define the reduction $\to$ as two Coq predicates: `red_expr` and `red_stat`. The first predicate has type `environment → heap → ext_expr → out_expr → Type` (and similarly for the statement reduction). The construction `ext_expr` refers to the extended syntax for expressions e_e. The inductive type `out_expr` is defined as being either the result of a terminating evaluation, containing a new environment, heap, and returned value, or an aborted evaluation, containing a new environment and heap.

```
1 Inductive out_expr :=
2   | out_expr_ter : environment → heap_o → value → out_expr
3   | out_expr_error : environment → heap_o → out_expr.
```

To ease the instrumentation, we directly add the annotations in the semantics: each rule of the semantics takes two additional arguments: the left-hand side annotation and the right-hand side annotation. However, there is no restriction on these annotations, we rely on the correctness properties of Section 3.5 to ensure they define the property of interest.

The semantics is thus parameterized by four types, corresponding to the left and right annotations for expressions and statements. These types are wrapped in a Coq record and used through projections such as `annot_e_l` (for left-hand-side annotations in expressions).

Figure 8 shows the rule for variables from this annotated semantics, where `ext_expr_expr` corresponds to the injection of expressions into extended expressions. The additional annotation arguments of type `annot_e_l` and `annot_e_r` are carried by every rule. As every rule contains such annotations, it is easy to write a function `extract_anot` taking such a derivation tree and returning the corresponding annotations. Every part of the Coq development that uses the reduction $\to$ but not the annotations (such as the interpreter) uses trivial annotations of unit type.

The annotations are then incrementally computed using Coq functions. Each of the new annotating passes takes the result of the previous pass as an argument to add its new annotations. The initial annotation is the trivial one, where every annotating types are unit. The definition of

$$\frac{E[\mathtt{x}] = v \qquad M[\mathtt{x}] = \tau_0}{\tau, M, d, E, H, \underline{\mathtt{x}} \to \tau', M, d \cup \{\mathtt{x}^{\tau_0}\}, E, H, v}\ \textsc{Var} \qquad \frac{H[l] = \bot \qquad H' = H[l \mapsto \underline{\{\}}] \qquad M' = M[l \mapsto \tau']}{\tau, M, d, E, H, \underline{\{\}} \to \tau', M, d \cup \left\{l^{\tau'}\right\}, E, H', l}\ \textsc{Obj}$$

$$\frac{H'[l] = o \qquad o[\mathtt{f}] = v \qquad M[(l, M[l], f)] = \tau_0}{\tau, M, d, E, H, \underline{(E', H', l)\ ._1 \mathtt{f}} \to \tau', M, d \cup \{(l, M[l], f)^{\tau_0}\}, E', H', v}\ \textsc{Fld1}$$

$$\frac{\tau_1, M, \emptyset, S, \underline{e} \to \tau_2, M', d, r \qquad \tau_3, M', \emptyset, S, \underline{\mathtt{if1}(r, s_1, s_2)} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{\mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2} \to \tau_5, M'', \emptyset, r'}\ \textsc{If}$$

$$\frac{\tau_1, M, \emptyset, S, e \to \tau_2, M', d, r \qquad \tau_3, M', \emptyset, S, \underline{\mathtt{while1}(r, x, s)} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{\mathtt{while}\ e\ \mathtt{do}\ s} \to \tau_5, M'', \emptyset, r'}\ \textsc{While}$$

$$\frac{\tau_1, M, \emptyset, S, e \to \tau_2, M', d, r \qquad \tau_3, M', d, S, \underline{\mathtt{x} =_1 r} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{\mathtt{x} = e} \to \tau_5, M'', \emptyset, r'}\ \textsc{Asg}$$

$$\frac{E' = E[\mathtt{x} \mapsto v] \qquad M' = M[\mathtt{x} \mapsto \tau']}{\tau, M, d, S, \underline{\mathtt{x} =_1 (E, H, v)} \to \tau', M', \emptyset, E', H}\ \textsc{Asg1}$$

$$\frac{\tau_1, M, \emptyset, S, e_1 \to \tau_2, M', d, r \qquad \tau_3, M', \emptyset, S, \underline{r.\mathtt{f} =_1 e_2} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{e_1.\mathtt{f} = e_2} \to \tau_5, M'', \emptyset, r'}\ \textsc{FldAsg}$$

$$\frac{\tau_1, M, \emptyset, S', e \to \tau_2, M', d, r \qquad \tau_3, M', d, S', \underline{l.\mathtt{f} =_2 r} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{(S', x).\mathtt{f} =_1 e} \to \tau_5, M'', \emptyset, r'}\ \textsc{FldAsg1}$$

$$\frac{H[l] = o \qquad o' = o[\mathtt{f} \mapsto v] \qquad H' = H'[l \mapsto o'] \qquad M' = M[(l, M[l], f) \mapsto \tau']}{\tau, M, d, S, \underline{l.\mathtt{f} =_2 (E, H, v)} \to \tau', M', \emptyset, E, H'}\ \textsc{FldAsg2}$$

$$\frac{\tau_1, M, \emptyset, S, e \to \tau_2, M', d, r \qquad \tau_3, M', \emptyset, S, \underline{\mathtt{delete1}\ r.\mathtt{f}} \to \tau_4, M'', \emptyset, r'}{\tau_0, M, \emptyset, S, \underline{\mathtt{delete}\ e.\mathtt{f}} \to \tau_5, M'', \emptyset, r'}\ \textsc{Delete}$$

Figure 6: Rules for Dependencies Annotations

annotations in our Coq development exactly follows the scheme presented in Section 3.2. This allows to only specify the parts of the analysis that effectively change their annotations, using a pattern matching construction ending with a Coq's wild card _ to deal with all the cases that just propagate the annotations. It has been written in a modular way, which is robust to changes. For example, a previous version of the annotations only modified partial traces on one side of the rules. As all the following annotating passes treat the traces as an abstract object whose type is parameterized, it was straightforward to update the Coq development to change traces on both sides of each rule.

Figure 9 shows the introduction of the last-modified annotation (see Figure 3c and 4c). This annotation is parameterized by another (traces for instance) here called `Locations`. In Coq, the heap M of Section 3.4 is represented by the record `LastChangeHeaps` defined on Line 4. Line 9 then states it is the left and right annotation types of this annotation. Next is the pattern matching defining the *axiom* rule for statement, and in particular the case of the assignment Line 17 which, as in Figure 4c, stores the current location τ in the annotation. Line 31 sums up the rules, stating that every rule

$$\frac{E' = E[\mathtt{x} \mapsto v] \qquad M' = M[\mathtt{x} \mapsto \tau']}{\tau, M, d, \Delta, S, \underline{\mathtt{x} =_1 (E, H, v)} \to \tau', M', \emptyset, \left\{\delta \hookleftarrow \mathtt{x}^{\tau'} | \delta \in d\right\} \cup \Delta, E', H} \textsc{Asg1}$$

$$\frac{H[l] = o \qquad o' = o[\mathtt{f} \mapsto v] \qquad H' = H'[l \mapsto o'] \qquad M' = M[(l, M[l], f) \mapsto \tau']}{\tau, M, d, \Delta, S, \underline{l.\mathtt{f} =_2 (E, H, v)} \to \tau', M', \emptyset, \left\{\delta \hookleftarrow (l, M[l], f)^{\tau'} | \delta \in d\right\} \cup \Delta, E, H'} \textsc{FldAsg2}$$

Figure 7: Rules for Annotating Dependencies of Statements

```
Inductive red_expr : environment →heap_o →ext_expr →out_expr → Type :=
  (* ... *)
  | red_expr_expr_var : annot_e_l Annots →annot_e_r Annots →
    ∀E H x v, getvalue E x v →red_expr E H (ext_expr_expr (expr_var x)) (out_expr E H v)
```

Figure 8: A Semantic Rule as Written in Coq

of this annotation just propagates their arguments, except the *axiom* rule for statements. As can be seen, the corresponding code is fairly short.

We have also defined an interpreter `run_expr : nat →environment →heap_o →expr →option out` taking as arguments an integer, an environment, a heap, and an expression and returning an output. The presence of a `while` in O'While allows the existence of non-terminating executions, whereas every Coq function must be terminating. To bypass this mismatch, the interpreters `run_expr` and `run_stat` (respectively running over expressions and statements) take an integer (the first argument of type `nat` above), called *fuel*. At each recursive call, this fuel is decremented, the interpreter giving up and returning `None` once it reaches 0. We have proven the interpreter is correct related to the semantics, and we have extracted it as an OCaml program using the Coq extraction mechanism.

4. Dependency Analysis

The annotating process makes the property we want to track appear explicitly in derivation trees. Our next step is to define an abstraction of the semantics for computing safe approximations of these properties, and to prove its correctness with respect to the instrumented semantics.

4.1. Abstract Domains

The analysis is expressed as a reduction relation operating over abstractions of the concrete semantic domains. The notion of program point will play a central role, as program points are used both in the abstraction of points of allocation and points of modification. This analysis thus uses the set PP of program points, so we assume that the input program is a result of Function Π defined in Appendix A.1. Property 4, defined in Appendix A.3, ensures that the added program points are correct with respect to the associated traces, which are used to name objects, and thus that this abstraction is sound. To avoid burdening notations, program points are only shown when needed.

Values are defined to be either basic values or locations. Regarding locations, we use the standard abstraction in which object locations are abstracted by the program points corresponding to the instruction that allocated the object. We abstract basic values, which are booleans in our setting, using a lattice $Bool^\sharp$. Thus $l^\sharp \in Loc^\sharp = \mathcal{P}(PP)$ and $v^\sharp \in Val^\sharp = Loc^\sharp + Bool^\sharp$. We define $v_l^\sharp$ as being

```
Variable Locations : Annotations.

Definition ModifiedAnnots := annot_s_r Locations.
Record LastModifiedHeaps : Type :=
  makeLastModifiedHeaps {
    LCEnvironment : heap var ModifiedAnnots;
    LCHeap : heap loc (heap prop_name ModifiedAnnots)}.

Definition LastModified := ConstantAnnotations LastModifiedHeaps.

Definition LastModifiedAxiom_s (r : LastModifiedHeaps)
  E H t o (R : red_stat Locations E H t o) :=
  let LCE := LCEnvironment r in
  let LCH := LCHeap r in
  let (_, tau) := extract_annot_s R in
  match R with
  | red_stat_ext_stat_assign_1 _ _ _ _ _ _ _ x _ _ ⇒
    let LCE' := write LCE x tau
    in makeLastModifiedHeaps LCE' LCH
  | red_stat_stat_delete _ _ _ _ _ l _ f _ _ _ _ _ _ ⇒
    let aob := read LCH l
    in let LCH' := write LCH l (write aob f tau)
    in makeLastModifiedHeaps LCE LCH'
  | red_stat_ext_stat_set_2 _ _ _ _ _ _ _ _ l _ f _ _ _ _ _ _ ⇒
    let aob := read LCH l
    in let LCH' := write LCH l (write aob f tau)
    in makeLastModifiedHeaps LCE LCH'
  | _ ⇒makeLastModifiedHeaps LCE LCH
  end.

Definition annotLastModified :=
  makeIterativeAnnotations LastModified
    (init_e Transmit) (axiom_e Transmit) (up_e Transmit) (down_e Transmit) (next_e Transmit)
    (up_s_e Transmit) (next_e_s Transmit)
    (init_s Transmit) LastModifiedAxiom_s (up_s Transmit) (down_s Transmit) (next_s Transmit).
```

Figure 9: Coq Definitions of the Last-Modified Annotation

either $v^\sharp$ if $v^\sharp \in Loc^\sharp$ or as $\emptyset$ otherwise.

For objects stored at heap locations, we keep trace of the values that the fields may reference. As with annotations, we record the last place each variable and field has been modified. Environments and heaps are thus abstracted as follows: $E^\sharp \in Env^\sharp = Var \to (\mathcal{P}(PP) \times Val^\sharp)$ maps variables to abstract values $v^\sharp$; $H^\sharp \in Heap^\sharp = Loc^\sharp \to Field \to (\mathcal{P}(PP) \times Val^\sharp)$ maps abstract locations to object abstractions (that map fields to abstract values), also storing their last place(s) of modification. Note that we shall freely use curryfied version of $H^\sharp \in Heap^\sharp$.

The two abstract domains inherit a lattice structure in the canonical way as monotone maps, ordered pointwise. The abstract heaps $H^\sharp$ map abstract locations $Loc^\sharp$ (sets of program points) to abstract object, but as locations are abstracted by sets, each write of a value $v^\sharp$ into an abstract heap at abstract location $l^\sharp$ implicitly yields a join between $v^\sharp$ and every value associated to an $l'^\sharp \sqsubseteq l^\sharp$. In practice, such an abstract heap $H^\sharp$ is implemented by a map from program points to abstract values and those writes yield joins with every $p \in l^\sharp$.

We abstract $l^\tau \in ALoc$ by the program point that allocated l^τ, and the traces by program points

(using $\prec$). We can thus abstract the relation $\Delta_\tau \in \mathit{Dep}$ (and the relation $\leadsto$) by making the natural abstraction $\Delta^\sharp$ of those definitions: $\Delta^\sharp \in \mathit{Dep}^\sharp = \mathcal{P}\left(\mathit{Source}^\sharp \times \mathit{Store}^\sharp\right)$; $\mathit{ALoc}^\sharp = \mathit{PP}$; $\mathit{Store}^\sharp = (\mathit{Var} \times \mathit{PP}) + (\mathit{PP} \times \mathit{Field} \times \mathit{PP})$; and $d^\sharp \in \mathit{Source}^\sharp = \mathit{PP} + \mathit{Store}^\sharp$. Abstract flows are written using the symbol $\leadsto^\sharp$. To avoid confusion, program points $p \in \mathit{PP}$ interpreted as elements of $\mathit{Source}^\sharp$ (thus representing locations) are written o^p.

Abstract flows are thus usual flows in which all traces have been replaced by program points. We've seen in Section 3.1 that there exists an abstraction relation $\prec$ between traces and program points such that $\tau \prec p$ if and only if p corresponds to the trace τ. This relation can be directly extended to Dep and $\mathit{Dep}^\sharp$: for instance for each $\mathtt{x}^\tau \in \mathit{Var} \times \mathit{Trace} \subset \mathit{Store}$ such that $\tau \prec p$, we have $\mathtt{x}^\tau \prec \mathtt{x}^p \in \mathit{Store}^\sharp$. Similarly, this relation $\prec$ can also be defined over Val and $\mathit{Val}^\sharp$, Env and $\mathit{Env}^\sharp$, and Heap and $\mathit{Heap}^\sharp$.

4.2. Abstract Reduction Relation

We formalize the analysis as an abstract reduction relation $\rightarrow^\sharp$ for expressions and statements: $E^\sharp, H^\sharp, \underline{s} \rightarrow^\sharp E'^\sharp, H'^\sharp, \Delta^\sharp$ and $E^\sharp, H^\sharp, \underline{e} \rightarrow^\sharp v^\sharp, d^\sharp$. On statements, the analysis returns an abstract environment, an abstract heap, and a partial dependency relation. On expressions, it returns the set of all its possible locations and the set of its dependencies. The analysis is correct if, for all statements, the result of the abstract reduction relation is a correct abstraction of the instrumented reduction. More precisely, the analysis is correct if for each statement $\underline{s}$ such that

$$\tau, M_\tau, d_\tau, \Delta_\tau, E_\tau, H_\tau, \underline{s} \rightarrow \tau', M_{\tau'}, d_{\tau'}, \Delta_{\tau'}, E_{\tau'}, H_{\tau'} \quad \text{and} \quad E^\sharp, H^\sharp, \underline{s} \rightarrow^\sharp E'^\sharp, H'^\sharp, \Delta^\sharp$$

where M_τ is chosen accordingly to E_τ and H_τ, $E_\tau \prec E^\sharp$, and $H_\tau \prec H^\sharp$, we have $\Delta_{\tau'} \prec \Delta^\sharp$. In other words, the analysis captures at least all the real flows, defined by the annotations.

Figure 10 shows the rules of this analysis. To avoid burdening notations, we denote by $d^\sharp \leadsto f$ the abstract dependency relation $\left\{(f_d, f) \middle| f_d \in d^\sharp\right\}$. Following the same scheme, we freely use the notation $l^\sharp.\mathtt{f}^p$ to denote the set $\left\{p_0.\mathtt{f}^p \middle| p_0 \in l^\sharp\right\}$. As an example, here is the rule for assignments:

$$\frac{E^\sharp, H^\sharp, \underline{e} \rightarrow^\sharp v^\sharp, d^\sharp}{E^\sharp, H^\sharp, \underline{\mathtt{x}^p = e} \rightarrow^\sharp E^\sharp\left[\mathtt{x} \mapsto \left(\{p\}, v^\sharp\right)\right], H^\sharp, \left(v_l^\sharp \cup d^\sharp\right) \leadsto^\sharp \mathtt{x}^p} \textsc{Asg}$$

This rule expresses that when encountering an assignment, an over-approximation of all the possible locations in the form of an abstract value $v^\sharp$ and of the dependencies $d^\sharp$ of the assigned expression $\underline{e}$ is computed. The abstract environment is then updated by setting the variable `x` to this new abstract value. Every possible flow from a potential dependency $\mathtt{y} \in d^\sharp$ or possible location value $v_l^\sharp$ of the expression $\underline{e}$ is marked as flowing into `x`. The position of `x` is taken into account in the resulting flows.

The Bin rule makes use of an abstract operation $\mathtt{op}^\sharp$, which depends on the operators added in the language. Figure 16 in Appendix B shows an example of analysis on the code we have seen on the previous sections, namely `x = {}; x.f = {}; if false then y = x.f else y = {}`.

There are several possible variations and extensions of this analysis. For one notable example it could be refined with strong updates on locations. For the moment, we leave for further work how exactly to annotate the semantics and to abstract locations in order to state whether or not an abstract location represents a unique concrete location in the heap.

4.3. Analysis in Coq

The abstract domains are essentially the same as the ones described in Section 4.1. They are straightforward to formalize as soon as basic constructions for lattices are available: the abstract domains are just specific instances of standard lattices from abstract interpretation (flat lattices,

$$\frac{}{E^\sharp, H^\sharp, \underline{\texttt{skip}} \to^\sharp E^\sharp, H^\sharp, \emptyset}\ \textsc{Skip} \qquad \frac{E^\sharp, H^\sharp, \underline{s_1} \to^\sharp E_1^\sharp, H_1^\sharp, \Delta_1^\sharp \qquad E_1^\sharp, H_1^\sharp, \underline{s_2} \to^\sharp E_2^\sharp, H_2^\sharp, \Delta_2^\sharp}{E^\sharp, H^\sharp, \underline{s_1; s_2} \to^\sharp E_2^\sharp, H_2^\sharp, \Delta_1^\sharp \cup \Delta_2^\sharp}\ \textsc{Seq}$$

$$\frac{E^\sharp, H^\sharp, \underline{s_1} \to^\sharp E_1^\sharp, H_1^\sharp, \Delta_1^\sharp \qquad E^\sharp, H^\sharp, \underline{s_2} \to^\sharp E_2^\sharp, H_2^\sharp, \Delta_2^\sharp}{E^\sharp, H^\sharp, \underline{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2} \to^\sharp E_1^\sharp \sqcup E_2^\sharp, H_1^\sharp \sqcup H_2^\sharp, \Delta_1^\sharp \cup \Delta_2^\sharp}\ \textsc{If}$$

$$\frac{E^\sharp \sqsubseteq E_0^\sharp \qquad H^\sharp \sqsubseteq H_0^\sharp \qquad E_0^\sharp, H_0^\sharp, \underline{s} \to^\sharp E_1^\sharp, H_1^\sharp, \Delta^\sharp \qquad E_1^\sharp \sqsubseteq E_0^\sharp \qquad H_1^\sharp \sqsubseteq H_0^\sharp}{E^\sharp, H^\sharp, \underline{\texttt{while } e \texttt{ do } s} \to^\sharp E_0^\sharp, H_0^\sharp, \Delta^\sharp}\ \textsc{While}$$

$$\frac{E^\sharp, H^\sharp, \underline{e} \to^\sharp v^\sharp, d^\sharp}{E^\sharp, H^\sharp, \underline{\texttt{x}^p \texttt{ = } e} \to^\sharp E^\sharp\left[\texttt{x} \mapsto \left(\{p\}, v^\sharp\right)\right], H^\sharp, \left(v_l^\sharp \cup d^\sharp\right) \leftsquigarrow^\sharp \texttt{x}^p}\ \textsc{Asg}$$

$$\frac{E^\sharp, H^\sharp, \underline{e_1} \to^\sharp v_1^\sharp, d_1^\sharp \qquad E^\sharp, H^\sharp, \underline{e_2} \to^\sharp v_2^\sharp, d_2^\sharp}{E^\sharp, H^\sharp, \underline{e_1\texttt{.f}^p \texttt{ = } e_2} \to^\sharp E^\sharp, H^\sharp \sqcup \left\{\left(v_1{}_l^\sharp, \texttt{f}\right) \mapsto \left(\{p\}, v_2^\sharp\right)\right\}, \left(v_2{}_l^\sharp \cup d_1^\sharp \cup d_2^\sharp\right) \leftsquigarrow^\sharp v_1{}_l^\sharp\texttt{.f}^p}\ \textsc{FldAsg}$$

$$\frac{E^\sharp, H^\sharp, \underline{e} \to^\sharp v^\sharp, d^\sharp \qquad H^\sharp[v_l^\sharp] \sqsubseteq o^\sharp \qquad o^\sharp[\texttt{f}] = \left(p_0, v_f^\sharp\right)}{E^\sharp, H^\sharp, \underline{\texttt{delete } e\texttt{.f}} \to^\sharp E^\sharp, H^\sharp, d^\sharp \leftsquigarrow^\sharp v_l^\sharp\texttt{.f}^{p_0}}\ \textsc{Del} \qquad \frac{}{E^\sharp, H^\sharp, \underline{c} \to^\sharp c^\sharp, \emptyset}\ \textsc{Cst}$$

$$\frac{E^\sharp[\texttt{x}] = \left(p_0, v^\sharp\right)}{E^\sharp, H^\sharp, \underline{\texttt{x}^p} \to^\sharp v^\sharp, \{\texttt{x}^{p_0}\}}\ \textsc{Var} \qquad \frac{E^\sharp, H^\sharp, \underline{e_1} \to^\sharp v_1^\sharp, d_1^\sharp \qquad E^\sharp, H^\sharp, \underline{e_2} \to^\sharp v_2^\sharp, d_2^\sharp}{E^\sharp, H^\sharp, \underline{e_1\ \texttt{op}\ e_2} \to^\sharp v_1^\sharp\ \texttt{op}^\sharp\ v_2^\sharp, d_1^\sharp \cup d_2^\sharp}\ \textsc{Bin}$$

$$\frac{}{E^\sharp, H^\sharp, \underline{\{\}^p} \to^\sharp \{p\}, \{o^p\}}\ \textsc{Obj} \qquad \frac{E^\sharp, H^\sharp, \underline{e} \to^\sharp v^\sharp, d^\sharp \qquad H^\sharp[v_l^\sharp] \sqsubseteq o^\sharp \qquad o^\sharp[\texttt{f}] = \left(p_0, v_f^\sharp\right)}{E^\sharp, H^\sharp, \underline{e\texttt{.f}^p} \to^\sharp v_f^\sharp, \left(v_l^\sharp\texttt{.f}^{p_0}\right) \cup d^\sharp}\ \textsc{Fld}$$

Figure 10: Rules for the Abstract Reduction Relation

power set lattices...). For the certification of lattices we refer to the Coq developments by David Pichardie [10].

Similarly to Section 3.6, the rules of the analyzer presented in Figure 10 are first defined as an inductive predicate of type

```
t AEnvironment → t AHeap → stat → t AEnvironment → t AHeap → t AFlows → Prop
```

where the two types `t AEnvironment` and `t AHeap` are the types of the abstract lattices for environments and heaps, and `t AFlows` the type of abstract flows, represented as a lattice for convenience. The analyzer is then defined by an extractable function of similar type (excepting the final "→ `Prop`"), the two definitions being proven equivalent. The situation for expressions is similar.

Once the analysis has been defined as well as the instrumentation, it's possible to formally prove the correctness of the abstract reduction rules with respect to the instrumentation. The property to prove is the one shown in Section 4.2: if from an empty heap, a program reduces to a heap E_τ, H_τ and flows Δ_τ, then if from the $\bot$ abstraction, a program reduces to $E^\sharp$, $H^\sharp$ and abstract flows $\Delta^\sharp$; that is, $[], \emptyset, [], \emptyset, \emptyset, \underline{s} \to \tau, M_\tau, \Delta_\tau, E_\tau, H_\tau$ and $\bot, \bot, \underline{s} \to^\sharp E^\sharp, H^\sharp, \Delta^\sharp$, then $E \prec E^\sharp$, $H \prec H^\sharp$ and $\Delta_\tau \prec \Delta^\sharp$. This is shown in [2].

5. Conclusion

Schmidt's natural semantics-based abstract interpretation is a rich framework which can be instantiated in a number of ways. In this paper, we have shown how the framework can be applied to the particular style of natural semantics called pretty big step semantics. We have studied a particular kind of intentional information about the program execution, *viz.*, how information flows from points of creation to points of use. This has lead us to define a particular abstraction of semantic derivation trees for describing points in the execution. This abstraction can then be further combined with other abstractions to obtain an abstract reduction relation that formalizes the static analysis.

Other systematic derivation of static analyses have taken small-step operational semantics as starting point. Cousot [4] has shown how to systematically derive static analyses for an imperative language using the principles of abstract interpretation. Midtgaard and Jensen [8, 9] used a similar approach for calculating control-flow analyses for functional languages from operational semantics in the form of abstract machines. Van Horn and Might [14] show how a series of analyses for functional languages can be derived from abstract machines. An advantage of using small-step semantics is that the abstract interpretation theory is conceptually simpler and more developed than its big-step counterpart. Our motivation for developing the big-step approach further is that the semantic framework has certain modularity properties that makes it a popular choice for formalizing real-sized programming languages.

Our preliminary experiments show that the semantics and its abstractions lend themselves well to being implemented in the Coq proof assistant. This is an important point, as some form of mechanization is required to evaluate the scalability of the method. Scalability is indeed one of the goals for this work. The present paper establishes the principles with which we hope to achieve the generation of an analysis for full JavaScript based on its Coq formalization. However, this will require some form of machine-assistance in the production of the abstract semantics. The present work provides a first experience of how to proceed. Further work will now have to extract the essence of this process and investigate how to program it in Coq.

One this has been achieved, we will be well armed to attack other analyses. One immediate candidate for further work is full information flow analysis, taking indirect flows due to conditionals into account. It would in particular be interesting to see if the resulting abstract semantics can be used for a rational reconstruction of the semantic foundations underlying the dynamic and hybrid information flow analysis techniques developed by Le Guernic, Banerjee, Schmidt and Jensen [7]. Combined with the extension to full JavaScript, this would provide a certified version of the recent information flow control mechanisms for JavaScript such as the monitor proposed by Hedin and Sabelfeld [6].

Bibliographie

[1] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. Jscert: Certified javascript. `http://jscert.org/`, 2012.

[2] M. Bodin, T. Jensen, and A. Schmitt. Pretty-big-step certified abstract interpretation, coq source code. `http://www.irisa.fr/celtique/aschmitt/research/owhileflows/`, 2013.

[3] A. Charguéraud. Pretty-big-step semantics. In *ESOP 2013*, pages 41–60. Springer, 2013.

[4] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[5] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *ISSTA 2011*, pages 177–187. ACM Press, 2011.

[6] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Proc. of the 25th Computer Security Foundations Symp. (CSF'12)*, pages 3–18. IEEE, 2012.

[7] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based Confidentiality Monitoring. In *ASIAN 2006*, pages 75–89. Springer LNCS vol. 4435, 2006.

[8] J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *SAS 2008*, volume 5079 of *LNCS*, pages 347–362. Springer Verlag, 2008.

[9] J. Midtgaard and T. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *ICFP 2009*, pages 287–298. ACM, 2009.

[10] D. Pichardie. Building certified static analysers by modular construction of well-founded lattices. In *FICS 2008*, volume 212 of *ENTCS*, pages 225–239, 2008.

[11] D. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). In *Proc. 2d Static Analysis Symposium (SAS'95)*, pages 1–18. Springer LNCS vol. 983, 1995.

[12] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, 2010.

[13] D. L. M. Valérie Gouranton. Dynamic slicing: a generic analysis based on a natural semantics format. *Journal of Logic and Computation*, 9(6), 1999.

[14] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP*, pages 51–62. ACM, 2010.

[15] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 42, 2007.

A. Traces and Program Points

In this section we define how to add program points to programs, in order to identify syntactic positions in the program, and we show how to relate traces from a semantic derivation with program points.

A.1. Program Points

Program points are defined as chains of atoms. The transformation Π described below takes a program point and a term, and annotates each sub-term with program points before and after the sub-term. The program point before a syntactic construct is a *context*, a chain of atoms indicating where to find the term in the initial program. The program point after is a context followed by the atom identifying the syntactic construct corresponding to the sub-term. For instance, the program point Seq2/Seq2/IfE refers to the point before `false` in the term `x = {}; x.f = {}; if false then y = x.f else y = {};` and Seq2/Seq2/IfE/Cst to the point after. The notion of "before" and "after" a program point is standard in data flow analysis, but is here given a semantics-based definition.

We write $\cdot$ for the empty program point, and we assume program points have a monoid structure, with / as concatenation and $\cdot$ as neutral element.

$$
\begin{aligned}
\Pi(\mathrm{PP}, \texttt{skip}) &= \mathrm{PP}, \texttt{skip}, \mathrm{PP/Skip} \\
\Pi(\mathrm{PP}, s_1; s_2) &= \mathrm{PP}, \Pi(\mathrm{PP/Seq1}, s_1); \Pi(\mathrm{PP/Seq2}, s_2), \mathrm{PP/Seq} \\
\Pi(\mathrm{PP}, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2) &= \mathrm{PP}, \texttt{if } \Pi(\mathrm{PP/IfE}, e) \texttt{ then } \Pi(\mathrm{PP/IfT}, s_1) \texttt{ else } \Pi(\mathrm{PP/IfF}, s_2), \mathrm{PP/If} \\
\Pi(\mathrm{PP}, \texttt{while } e \texttt{ do } s) &= \mathrm{PP}, \texttt{while } \Pi(\mathrm{PP/WhileE}, e) \texttt{ do } \Pi(\mathrm{PP/WhileS}, s), \mathrm{PP/While} \\
\Pi(\mathrm{PP}, \texttt{x = } e) &= \mathrm{PP}, \texttt{x = } \Pi(\mathrm{PP/AsgE}, e), \mathrm{PP/Asg} \\
\Pi(\mathrm{PP}, e_1\texttt{.f = } e_2) &= \mathrm{PP}, \Pi(\mathrm{PP/FldAsgL}, e_1)\texttt{.f = } \Pi(\mathrm{PP/FldAsgV}, e_2), \mathrm{PP/FldAsg} \\
\Pi(\mathrm{PP}, \texttt{delete } e\texttt{.f}) &= \mathrm{PP}, \texttt{delete } \Pi(\mathrm{PP/DelE}, e)\texttt{.f}, \mathrm{PP/Del} \\
\Pi(\mathrm{PP}, c) &= \mathrm{PP}, c, \mathrm{PP/Cst} \\
\Pi(\mathrm{PP}, \texttt{x}) &= \mathrm{PP}, \texttt{x}, \mathrm{PP/Var} \\
\Pi(\mathrm{PP}, e_1 \texttt{ op } e_2) &= \mathrm{PP}, \Pi(\mathrm{PP/BinL}, e_1) \texttt{ op } \Pi(\mathrm{PP/BinR}, e_2), \mathrm{PP/Bin} \\
\Pi(\mathrm{PP}, \texttt{\{\}}) &= \mathrm{PP}, \texttt{\{\}}, \mathrm{PP/Obj} \\
\Pi(\mathrm{PP}, e\texttt{.f}) &= \mathrm{PP}, \Pi(\mathrm{PP/FldE}, e)\texttt{.f}, \mathrm{PP/Fld}
\end{aligned}
$$

To derive program points from traces, we first need to define an operator that deletes atoms in a chain up to a given atom. More precisely, the operator $\mathrm{PP} \downarrow_{\mathrm{Name}}$ removes the shortest suffix of PP that starts with Name, included. Formally, it is defined as follows.

$$
\begin{aligned}
\cdot \downarrow_{\mathrm{Name}} &= \cdot \\
\mathrm{PP/Name} \downarrow_{\mathrm{Name}} &= \mathrm{PP} \\
\mathrm{PP/Name'} \downarrow_{\mathrm{Name}} &= \mathrm{PP} \downarrow_{\mathrm{Name}} \quad \text{if Name'} \neq \mathrm{Name}
\end{aligned}
$$

A.2. Traces

We give in Figure 11 the names that are added to the trace for each rule, following the general annotation scheme of Section 3.2. In the *next* step, the annotation of the left of the current rule is ignored, only the annotation on the right of the first premise is used.

Rule	*axiom*	*up*	*next*	*down*
Skip	Skip			
Seq		Seq1	$\overline{\text{Seq1}}$	Seq
Seq1		Seq2		$\overline{\text{Seq2}}$
If		IfE	$\overline{\text{IfE}}$	If
IfTrue		IfT		$\overline{\text{IfT}}$
IfFalse		IfF		$\overline{\text{IfF}}$
While		WhE	$\overline{\text{WhE}}$	While
WhileTrue1		WhS	$\overline{\text{WhS}}$	WhT1
WhileTrue2		WhL		WhT2
WhileFalse	WhF			
Asg		AsgE	$\overline{\text{AsgE}}$	Asg
Asg1	Asg1			
FldAsg		FldAsgL	$\overline{\text{FldAsgL}}$	FldAsg
FldAsg1		FldAsgV	$\overline{\text{FldAsgV}}$	FldAsg1
FldAsg2	FldAsg2			
Del		DelE	$\overline{\text{DelE}}$	Del
Del1	Del1			
Cst	Cst			
Var	Var			
Bin		BinL	$\overline{\text{BinL}}$	Bin
Bin1		BinR	$\overline{\text{BinR}}$	Bin1
Bin2	Bin2			
Obj	Obj			
Fld		FldE	$\overline{\text{FldE}}$	Fld
Fld1	Fld1			
Abort	Abort			

Figure 11: Traces Definition

A.3. From Traces to Program Points

We next define a function $\mathcal{T}$ from traces to program points. There are two challenges in doing so. First, traces mention everything that has happened up to the point under consideration. Program points, however, hide everything that is not under the current execution context. We take care of this folding using the deletion operator defined above. The second challenge is to decide what program point to assign for extended statements and expressions, and when to decide that a statement has finished executing. To illustrate this challenge, we consider the case of a while loop, where in the initial environment we have `x` and `y` equal to `true` and z equal to `false`.

The evaluation of the variable `x` only adds Var at the end of the trace. The evaluation of the two assignments appends the following sequence to the trace, which we call τ_s in the following.

$$[\text{Seq1}; \text{AsgE}; \text{Var}; \overline{\text{AsgE}}; \text{Asg1}; \text{Asg}; \overline{\text{Seq1}}; \text{Seq2}; \text{AsgE}; \text{Var}; \overline{\text{AsgE}}; \text{Asg1}; \text{Asg}; \overline{\text{Seq2}}; \text{Seq}]$$

$$
\dfrac{S,\texttt{x} \to S,\texttt{true} \qquad \dfrac{S,\texttt{x = y; y = z} \to S' \qquad \dfrac{\dfrac{S',\texttt{x} \to S',\texttt{true} \qquad \dfrac{S',\texttt{x = y; y = z} \to S'' \qquad \dfrac{\dfrac{S'',\texttt{x} \to S'',\texttt{false} \qquad \textcircled{2}\ S'',\texttt{while1}((S'',\texttt{false}),\texttt{x},\texttt{x = y; y = z}) \to S''\ \textcircled{1}}{S'',\texttt{while x do x = y; y = z} \to S''\ \textcircled{1}}}{\textcircled{3}\ S',\texttt{while2}(S'',\texttt{x},\texttt{x = y; y = z}) \to S''\ \textcircled{1}}}{\textcircled{2}\ S',\texttt{while1}((S',\texttt{true}),\texttt{x},\texttt{x = y; y = z}) \to S''\ \textcircled{1}}}{S',\texttt{while x do x = y; y = z} \to S''\ \textcircled{1}}}{\textcircled{3}\ S,\texttt{while2}(S',\texttt{x},\texttt{x = y; y = z}) \to S''\ \textcircled{1}}}{\textcircled{2}\ S,\texttt{while1}((S,\texttt{true}),\texttt{x},\texttt{x = y; y = z}) \to S''\ \textcircled{1}}}{S,\texttt{while x do x = y; y = z} \to S''\ \textcircled{1}}
$$

Figure 12: Running a While loop

The whole trace at the end of the execution has the following form.

$$
\begin{aligned}
&[\textsc{WhE}; \textsc{Var}; \overline{\textsc{WhE}}; \textsc{WhS}] +\!\!+\ \tau_s +\!\!+ [\overline{\textsc{WhS}}; \textsc{WhL}] +\!\!+ \\
&\qquad [\textsc{WhE}; \textsc{Var}; \overline{\textsc{WhE}}; \textsc{WhS}] +\!\!+\ \tau_s +\!\!+ [\overline{\textsc{WhS}}; \textsc{WhL}] +\!\!+ \\
&\qquad\qquad [\textsc{WhE}; \textsc{Var}; \overline{\textsc{WhE}}; \textsc{WhF}] +\!\!+ \\
&\qquad\qquad\qquad [\textsc{While}; \textsc{WhT2}; \textsc{WhT1}; \textsc{While}; \textsc{WhT2}; \textsc{WhT1}; \textsc{While}]
\end{aligned}
$$

The program point on the right-hand-side of every $\to S''$ should be WHILE (①), as it corresponds to the end of the program. The program point before every `while1` (②) should be the same on as right after evaluating the condition, namely WHILEE/VAR. Similarly, the one before `while2` (③) is taken to be the one right after finishing to evaluate the sequence, namelyWHILES/SEQ.

Following this intuition, we define in Figure 13 the $\mathcal{T}$ function that takes a trace and an already computed program point, then creates a program point. A quasi invariant is that part of a program point is deleted only if a new atom is added, reflecting the notion that evaluation never goes back in the program syntax. There is a crucial exception, though: when doing a loop for a `while` loop (premise of rule WHILE2), then the program point jumps back to right before the while loop.

We now state that program points can correctly be extracted from traces. To this end, we consider a derivation where the terms contain program points.

Property 4 *Let t be a term and $t' = \Pi(\cdot, t)$. For any occurrence of a rule the form $\tau, S, PP, t, PP' \to \tau', r$ in any annotated derivation tree from t', we have $PP = \mathcal{T}(\tau, \cdot)$, $PP' = \mathcal{T}(\tau', \cdot)$.*

$$
\begin{array}{rl}
\mathcal{T}([], \mathrm{PP}) = \mathrm{PP} & \\
\mathcal{T}(\textsc{Skip} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{Skip}) & \\
\mathcal{T}(\textsc{Seq1} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{Seq1}) & \mathcal{T}(\overline{\textsc{Seq1}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\tau, \mathrm{PP}/\textsc{Seq}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{Seq2} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{Seq1}} /\textsc{Seq2}) & \mathcal{T}(\overline{\textsc{Seq2}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{Seq2}} /\textsc{Seq}) \\
\mathcal{T}(\textsc{IfE} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{IfE}) & \mathcal{T}(\overline{\textsc{IfE}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{If} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{IfT} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{IfE}} /\textsc{IfT}) & \mathcal{T}(\overline{\textsc{IfT}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{IfT}} /\textsc{If}) \\
\mathcal{T}(\textsc{IfF} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{IfE}} /\textsc{IfF}) & \mathcal{T}(\overline{\textsc{IfF}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{IfF}} /\textsc{If}) \\
\mathcal{T}(\textsc{WhE} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{WhE}) & \mathcal{T}(\overline{\textsc{WhE}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{While} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{WhS} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{WhE}} /\textsc{WhS}) & \mathcal{T}(\overline{\textsc{WhS}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{WhT1} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{WhL} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{WhS}}) & \\
\mathcal{T}(\textsc{WhT2} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{WhF} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{While}) & \\
\mathcal{T}(\textsc{AsgE} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{Asge}) & \mathcal{T}(\overline{\textsc{AsgE}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Asg} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{Asg1} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{AsgE}} /\textsc{Asg}) & \\
\mathcal{T}(\textsc{FldAsgL} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{FldAsgL}) & \mathcal{T}(\overline{\textsc{FldAsgL}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{FldAsg} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{FldAsgV} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{FldAsgL}} /\textsc{FldAsgV}) & \mathcal{T}(\overline{\textsc{FldAsgV}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{FldAsg1} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{FldAsg2} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{FldAsgV}} /\textsc{FldAsg}) & \\
\mathcal{T}(\textsc{DelE} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{DelE}) & \mathcal{T}(\overline{\textsc{DelE}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Del} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{Del1} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{DelE}} /\textsc{Del}) & \\
\mathcal{T}(\textsc{Cst} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{Cst}) & \\
\mathcal{T}(\textsc{Var} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{Var}) & \\
\mathcal{T}(\textsc{BinL} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{BinL}) & \mathcal{T}(\overline{\textsc{BinL}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Bin} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{BinR} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{BinL}} /\textsc{BinR}) & \mathcal{T}(\overline{\textsc{BinR}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Bin1} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{Bin2} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{BinR}} /\textsc{Bin}) & \\
\mathcal{T}(\textsc{Obj} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{Obj}) & \\
\mathcal{T}(\textsc{FldE} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}/\textsc{FldE}) & \mathcal{T}(\overline{\textsc{FldE}} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) \\
\mathcal{T}(\textsc{Fld} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP}) & \\
\mathcal{T}(\textsc{Fld1} :: \tau, \mathrm{PP}) = \mathcal{T}(\tau, \mathrm{PP} \downarrow_{\textsc{FldE}} /\textsc{Fld}) & \\
\mathcal{T}(\textsc{Abort} :: \tau, \mathrm{PP}) = \mathrm{PP} &
\end{array}
$$

Figure 13: Traces to Program Points

B. Derivation examples

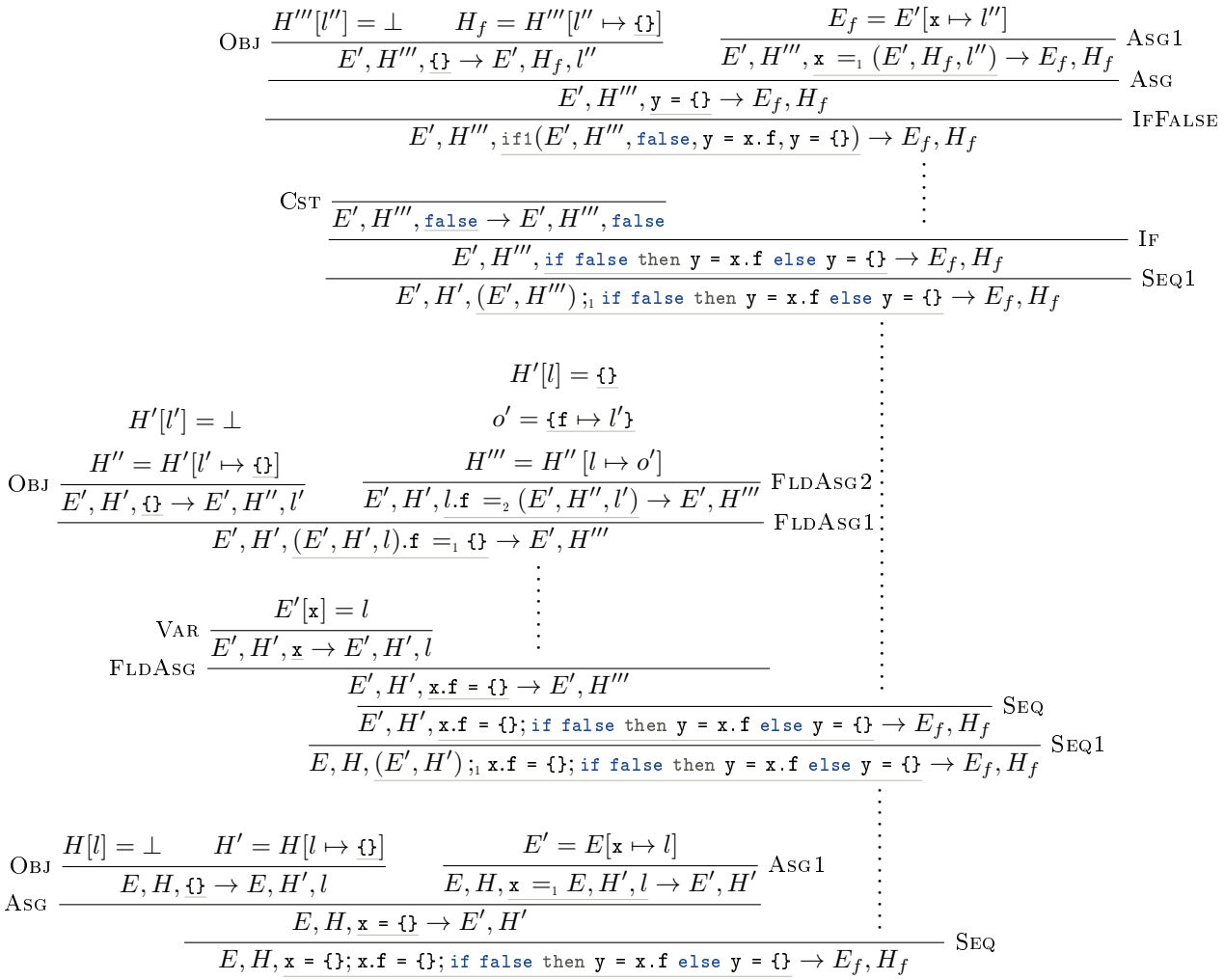


Figure 14: Pretty-big-step derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{E, H, \texttt{true} \to E, H, \texttt{true}}\,\textsc{Cst}}{E, H, \texttt{x = true} \to E', H}\,\textsc{Asg} \qquad \dfrac{E' = E[\texttt{x} \mapsto \texttt{true}]}{E, H, \texttt{x} =_1 (E, H, \texttt{true}) \to E', H}\,\textsc{Asg1}}{E, H, \texttt{x = true; y = x} \to E'', H}\,\textsc{Seq} \qquad \vdots}{}$$

$$\dfrac{\dfrac{\dfrac{E'[\texttt{x}] = \texttt{true}}{E', H, x \to E', H, \texttt{true}}\,\textsc{Var} \qquad \dfrac{E'' = E'[\texttt{y} \mapsto \texttt{true}]}{E', H, \texttt{x} =_1 E', H, \texttt{true} \to E'', H}\,\textsc{Asg1}}{E', H, \texttt{y = x} \to E'', H}\,\textsc{Asg}}{E, H, (E', H) ;_1 \texttt{y = x} \to E'', H}\,\textsc{Seq1}$$

(a) Unannotated Derivation

$$\tau_1 = [] \qquad \tau_2 = [\textsc{Seq1}] \qquad \tau_3 = \tau_2 +\!\!+ [\textsc{AsgE}] \qquad \tau_4 = \tau_3 +\!\!+ [\textsc{Cst}] \qquad \tau_5 = \tau_4 +\!\!+ [\overline{\textsc{AsgE}}]$$

$$\tau_6 = \tau_5 +\!\!+ [\textsc{Asg1}] \qquad \tau_7 = \tau_6 +\!\!+ [\textsc{Asg}] \qquad \tau_8 = \tau_7 +\!\!+ [\overline{\textsc{Seq1}}] \qquad \tau_9 = \tau_8 +\!\!+ [\textsc{Seq2}] \qquad \tau_{10} = \tau_9 +\!\!+ [\textsc{AsgE}]$$

$$\tau_{11} = \tau_{10} +\!\!+ [\textsc{Var}] \qquad \tau_{12} = \tau_{11} +\!\!+ [\overline{\textsc{AsgE}}] \qquad \tau_{13} = \tau_{12} +\!\!+ [\textsc{Asg1}] \qquad \tau_{14} = \tau_{13} +\!\!+ [\textsc{Asg}]$$

$$\tau_{15} = \tau_{14} +\!\!+ [\overline{\textsc{Seq2}}] \qquad \tau_{16} = \tau_{15} +\!\!+ [\textsc{Seq}]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\tau_3, E, H, \texttt{true} \to \tau_4, E, H, \texttt{true}}\,\textsc{Cst}}{\tau_2, E, H, \texttt{x = true} \to \tau_7, E', H}\,\textsc{Asg} \qquad \dfrac{E' = E[\texttt{x} \mapsto \texttt{true}]}{\tau_5, E, H, \texttt{x} =_1 (E, H, \texttt{true}) \to \tau_6, E', H}\,\textsc{Asg1}}{\tau_1, E, H, \texttt{x = true; y = x} \to \tau_{16}, E'', H}\,\textsc{Seq} \qquad \vdots}{}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{E'[\texttt{x}] = \texttt{true}}{\tau_{10}, E', H, x \to \tau_{11}, E', H, \texttt{true}}\,\textsc{Var} \qquad \dfrac{E'' = E'[\texttt{y} \mapsto \texttt{true}]}{\tau_{12}, E', H, \texttt{x} =_1 E', H, \texttt{true} \to \tau_{13}, E'', H}\,\textsc{Asg1}}{\tau_9, E', H, \texttt{y = x} \to \tau_{14}, E'', H}\,\textsc{Asg}}{\tau_8, E, H, (E', H) ;_1 \texttt{y = x} \to \tau_{15}, E'', H}\,\textsc{Seq1}$$

(b) Derivation Annotated With Traces

Figure 15: Annotating A Simple Derivation

$$E_1^\sharp = \{\mathtt{x} \mapsto (\{p_1\}, \{p_2\})\} \qquad E_2^\sharp = \{\mathtt{x} \mapsto (\{p_1\}, \{p_2\}), \mathtt{y} \mapsto (\{p_6\}, \{p_5\})\}$$

$$E_3^\sharp = \{\mathtt{x} \mapsto (\{p_1\}, \{p_2\}), \mathtt{y} \mapsto (\{p_9\}, \{p_{10}\})\} \qquad E_4^\sharp = \{\mathtt{x} \mapsto (\{p_1\}, \{p_2\}), \mathtt{y} \mapsto (\{p_6, p_9\}, \{p_5, p_{10}\})\}$$

$$H_1^\sharp = \{(p_2, \mathtt{f}) \mapsto (\{p_4\}, \{p_5\})\}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E_1^\sharp[\mathtt{x}] = (\{p_1\}, \{p_2\})}{E_1^\sharp, H_1^\sharp, \mathtt{x}^{p_7} \to^\sharp \{p_2\}, \{\mathtt{x}^{p_1}\}} \text{Var} \qquad H_1^\sharp[\{o^{p_2}\}][\mathtt{f}] = (\{p_4\}, \{p_5\})}{E_1^\sharp, H_1^\sharp, \mathtt{x}^{p_7}.\mathtt{f}^{p_8} \to^\sharp \{p_5\}, \{o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\}} \text{Fld} \qquad \vdots}{E_1^\sharp, H_1^\sharp, \mathtt{y}^{p_6} = \mathtt{x}.\mathtt{f} \to^\sharp E_2^\sharp, H_1^\sharp, \{\{o^{p_5}, o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\} \hookleftarrow \mathtt{y}^{p_6}\}} \text{Asg} \qquad \vdots}{E_1^\sharp, H_1^\sharp, \mathtt{if\ false\ then\ y = x.f\ else\ y = \{\}} \to^\sharp E_4^\sharp, H_1^\sharp, \{\{o^{p_5}, o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\} \hookleftarrow \mathtt{y}^{p_6}, o^{p_{10}} \hookleftarrow \mathtt{y}^{p_9}\}} \text{If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{E_1^\sharp[\mathtt{x}] = (\{p_1\}, \{p_2\})}{E_1^\sharp, \bot, \mathtt{x}^{p_3} \to^\sharp \{p_2\}, \{\mathtt{x}^{p_1}\}} \text{Var} \qquad \dfrac{}{E_1^\sharp, \bot, \{\}^{p_5} \to^\sharp \{p_5\}, \{o^{p_5}\}} \text{Obj}}{E_1^\sharp, \bot, \mathtt{x}^{p_3}.\mathtt{f}^{p_4} = \{\}^{p_5} \to^\sharp E_1^\sharp, H_1^\sharp, \{\{\mathtt{x}^{p_1}, o^{p_5}\} \hookleftarrow o^{p_2}.\mathtt{f}^{p_4}\}} \text{FldAsg} \qquad \vdots}{E_1^\sharp, \bot, \mathtt{x.f = \{\};if\ false\ then\ y = x.f\ else\ y = \{\}} \to^\sharp E_4^\sharp, H_1^\sharp, \{\{\mathtt{x}^{p_1}, o^{p_5}\} \hookleftarrow o^{p_2}.\mathtt{f}^{p_4}, \{o^{p_5}, o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\} \hookleftarrow \mathtt{y}^{p_6}, o^{p_{10}} \hookleftarrow \mathtt{y}^{p_9}\}} \text{Seq}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\bot, \bot, \{\}^{p_2} \to^\sharp \{p_2\}, \{o^{p_2}\}} \text{Obj}}{\bot, \bot, \mathtt{x}^{p_1} = \{\}^{p_2} \to^\sharp E_1^\sharp, \bot, \{o^{p_2} \hookleftarrow \mathtt{x}^{p_1}\}} \text{Asg} \qquad \vdots}{\bot, \bot, \mathtt{x = \{\}}^{p_2}\mathtt{;x.f = \{\}}^{p_5}\mathtt{;if\ false\ then\ y = x.f\ else\ y = \{\}}^{p_{10}} \to^\sharp E_4^\sharp, H_1^\sharp, \left\{\begin{array}{l} o^{p_2} \hookleftarrow \mathtt{x}^{p_1}, \{\mathtt{x}^{p_1}, o^{p_5}\} \hookleftarrow o^{p_2}.\mathtt{f}^{p_4}, \\ \{o^{p_5}, o^{p_2}.\mathtt{f}^{p_8}, \mathtt{x}^{p_1}\} \hookleftarrow \mathtt{y}^{p_6}, o^{p_{10}} \hookleftarrow \mathtt{y}^{p_9} \end{array}\right\}} \text{Seq}$$

Figure 16: Analysis Example

Comment un chameau peut-il écrire un journal ?

Julien Signoles[1]

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

Résumé

Dans *Frama-C*, plate-forme d'analyse de code *C* développée en *OCaml*, un journal est un script *OCaml* généré automatiquement et permettant de reproduire les actions utilisateurs, notamment effectuées *via* l'interface utilisateur. Outre la reproductibilité des résultats qui est nécessaire dans un contexte industriel soumis à des exigences de certification fortes comme la norme avionique DO-178C, un journal permet d'automatiser le pilotage de l'outil dans un contexte d'utilisation particulier. Cet article présente comment le mécanisme de génération du journal de *Frama-C*, appelé *journalisation* et requérant intrinsèquement de l'introspection, a été développé en *OCaml*, en combinant typage statique et dynamique.

1. Introduction

Frama-C [4] est une plate-forme d'analyse de code *C* développée en *OCaml* ayant vocation à être – et étant déjà – utilisée pour obtenir des garanties sur des programmes embarqués critiques dans un contexte industriel soumis à des contraintes de certification fortes, comme la norme avionique DO-178C [5]. Dans ce contexte, les outils d'analyse de code doivent fournir un moyen de reproduire les résultats obtenus lors d'une session utilisateur. Cette reproductibilité permet notamment de rejouer facilement les suites de tests et de tracer plus facilement la provenance des résultats obtenus (en particulier en cas de mauvais résultats, par exemple s'ils sont trop imprécis ou, pire, incorrects).

Pour satisfaire cette exigence, *Caveat* [2], l'ancêtre de *Frama-C* notamment utilisé lors du développement de l'A380, possède une notion de *journal*. Un journal est un script, ici en *OCaml*, généré automatiquement et permettant de reproduire toutes les actions utilisateurs, notamment effectuées au travers de l'interface utilisateur (GUI). Outre la reproductibilité requise par les processus de certification, un journal possède également l'avantage de permettre à l'utilisateur de créer facilement des scripts pour paramétrer l'outil. Automatiser ce pilotage de l'outil permet de gagner un temps important et, donc, de réduire les coûts. Grâce au journal, il est possible de réaliser ce processus une première fois manuellement sur un exemple précis, de récupérer le journal automatiquement généré, de le modifier éventuellement légèrement (pour, par exemple, le généraliser un peu), avant de l'exécuter autant de fois que nécessaire sur différentes applications. Le fait que le code *OCaml* du journal soit généré automatiquement permet, en outre, de réduire l'expertise requise à son écriture, tout particulièrement en *OCaml* et en connaissance des interfaces programmatiques (APIs) de l'outil. Ceci nécessite néanmoins que le code généré soit humainement lisible et raisonnablement détaillé.

Ces avantages de reproductibilité et d'automatisation demeurent en dehors de toute utilisation industrielle de l'outil. Par exemple, la reproductibilité permet d'obtenir de meilleurs rapports de bogues en provenance des utilisateurs car, en son absence, il est parfois difficile à ces derniers de décrire comment reproduire une erreur observée dans la GUI, à la suite d'un enchaînement d'événements consécutifs imprévus et survenus un peu par hasard. L'automatisation permet, quant à elle, de combiner facilement plusieurs analyses dans la GUI, pour en obtenir une nouvelle, plus puissante,

à faible coût. C'est tout particulièrement vrai pour *Frama-C*, fondé sur une architecture collaborative dans laquelle chaque analyseur est un greffon particulier pouvant notamment exploiter les résultats des autres, le noyau de *Frama-C* étant chargé de consolider les résultats globaux [3].

Pour ces différentes raisons, il est justifié de développer une notion de journal similaire à celle de *Caveat* au sein de *Frama-C*. Néanmoins, alors que *Frama-C* est développé en *OCaml*, le mécanisme de génération du journal dans *Caveat* – appelé *journalisation* – est programmé en *C++* et brise la sûreté du typage apportée par ce langage en abusant du transtypage (autrement dit, des *casts*). Le but de cet article est de montrer comment il a été possible d'obtenir une fonctionnalité similaire dans un langage à typage statique fort comme *OCaml*, en combinant typage statique et dynamique. L'implantation utilise en particulier, certes à faibles doses, types fantômes, types localement abstraits, types algébriques généralisés (GADTs) et récursion polymorphe. Par ailleurs, nous ne connaissons pas d'autres outils, en *OCaml* ou non, présentant un système similaire.

La section 2 offre une vue générale de la journalisation dans *Frama-C*. La section 3 présente les possibilités de typage dynamique de *Frama-C* utiles à notre étude. Enfin, la section 4 détaille comment ce mécanisme est implémenté.

2. Vue générale de la journalisation

Cette section explique les spécifications attendues de la journalisation (section 2.1), puis leurs déclinaisons dans le contexte de *Frama-C* (section 2.2), avant de présenter les principes de ce mécanisme au sein de cette plateforme (section 2.3) et de détailler un exemple (section 2.4).

2.1. Spécification informelle

Lors de l'exécution d'un logiciel, en particulier s'il bénéficie d'une GUI, un certain nombre d'actions sont entreprises par l'utilisateur, chacune d'elles déclenchant une série d'opérations internes engendrant un certains nombres d'effets dont certains sont présentés à l'utilisateur. Le but de la journalisation est d'instrumenter cette exécution, de manière à pouvoir générer un programme, appelé journal, reproduisant cette série d'opérations internes et, donc, engendrant les mêmes résultats.

Cette spécification générale de la journalisation permet d'en comprendre l'objectif global. Elle est néanmoins informelle et succincte et laisse ainsi la place à une liberté d'interprétation qu'il convient de préciser au cas par cas : qu'entend-on en particulier par *« opérations internes »*, *« effets »* et *« résultats observés »* ? La liberté laissée est nécessaire pour atteindre l'objectif initial qui est, rappelons-le, de pouvoir reproduire les résultats observés : l'exécution du journal doit aboutir à un résultat observationnellement équivalent à l'exécution dont il est issu, mais cette notion d'observabilité (*i.e.* d'équivalence observationnelle) dépend du contexte. Il est en effet peu probable que ce soit la même pour un programme encodant un protocole réseau échangeant des messages, pour un éditeur de texte ou pour un analyseur de code *C*.

Ce cadre général est néanmoins suffisant pour exprimer trois prérequis à remplir pour que l'exécution d'un journal J soit bien observationnellement équivalente à une exécution donnée d'un programme P dans un environnement d'utilisation particulier. D'abord, les effets produits par P doivent pouvoir l'être par J *(propriété P1)*, le moyen le plus simple pour cela étant que J ait accès à l'API de P. Ensuite, les résultats observés ne doivent dépendre que des effets : « mêmes effets, mêmes résultats » pourrait-on dire *(P2)*. Pour y parvenir, il faut que P et J soient déterministes, ce qui implique par exemple de fixer une même graine stable au générateur de nombres aléatoires ou de ne pas avoir plusieurs ordonnancements de processus possibles (à moins qu'un seul processus concerné n'ait un impact sur les résultats). Enfin, il faut disposer d'un état initial de J et d'un état initial de P équivalents avant d'entreprendre la première opération à écrire dans le journal *(P3)*, cette notion d'équivalence d'états dépendant de l'équivalence observationnelle préalablement mentionnée.

2.2. Contexte : *Frama-C*

Pour rendre le discours plus concret, plaçons nous désormais dans le cadre de *Frama-C*. Cet outil est une plate-forme extensible d'analyse de code *C* développée en *OCaml* (et uniquement en *OCaml* : l'emploi de *Camlp4* ou *Camlp5* est notamment prohibé) [5]. L'extensibilité est obtenue grâce à une architecture modulaire fondée sur des greffons pouvant communiquer entre eux *via* un noyau central. Elle est schématisée figure 1. Le noyau fournit également, d'une part, l'arbre de syntaxe abstraite du code *C*, annoté en *ACSL* [1], qui doit être analysé et, d'autre part, un certain nombre de services de base, sous forme d'autant de bibliothèques dédiées, dont le journal fait parti.

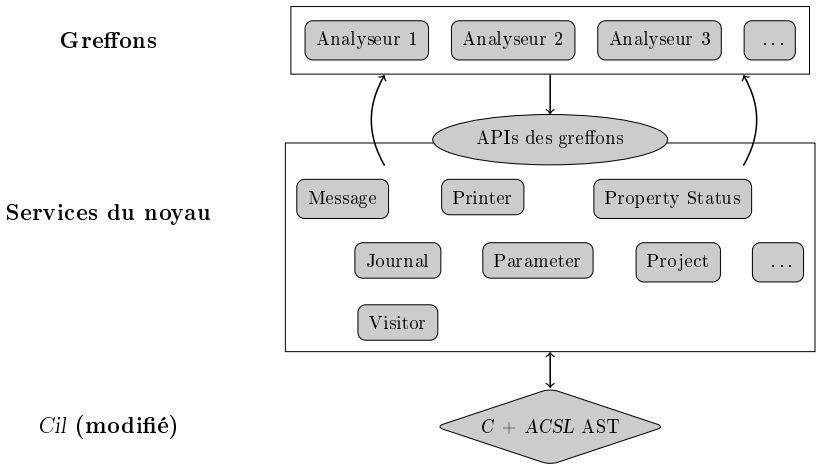


FIGURE 1 – Vue fonctionnelle de *Frama-C*.

Les greffons sont des programmes *OCaml*, chargés le plus souvent dynamiquement, ayant accès aux APIs du noyau et à celles des autres greffons [9]. Ce sont en général des analyseurs de code (analyse de valeurs par interprétation abstraite, preuve de programmes par calcul de plus faible précondition, transformations de programmes dédiées à la vérification de propriétés temporelles ou à la vérification d'annotations à l'exécution, ...) [4], mais rien n'empêche par exemple de programmer un serveur web comme un greffon *Frama-C*[1]. Il est également possible de développer des greffons légers, appelés *scripts*, qui requièrent un peu moins de développement (pas de `Makefile`, pas de pré-compilation nécessaire, ...) au prix de fonctionnalités légèrement bridées (un seul fichier *OCaml*, ...). Ces scripts sont parfaits pour développer de petites extensions de la plateforme, pour conduire des expérimentations ou piloter rapidement et automatiquement des analyses.

Dans ce contexte, il est apparu naturel de proposer à l'utilisateur le journal comme un script *Frama-C*. En effet, les trois prérequis *P1*, *P2* et *P3* mentionnés préalablement sont alors remplis : le script a accès aux APIs de *Frama-C (P1)*, *Frama-C* garantit que les mêmes effets conduisent aux mêmes résultats *(P2)* et sa phase d'initialisation, quoique complexe, assure que le chargement du script interviendra au moment adéquat pour être dans le bon état initial *(P3)*. Nous reviendrons néanmoins plus précisément sur ces points ci-après.

Nous pouvons maintenant clarifier pour *Frama-C* et son langage support *OCaml* ce que nous entendons par *« opérations internes »*, *« effets »* et *« résultats observés »* dont nous avons discutés en section 2.1. Les opérations internes sont des fonctions *OCaml* exportées par l'API de *Frama-C*, de manière à pouvoir être appelées par le journal et à garantir *P1*. Les résultats observés sont ceux affichés sur la sortie standard lorsque *Frama-C* est lancé en mode console (ils sont aussi présents

1. D'ailleurs, cela a déjà été réalisé pour expérimenter une GUI de *Frama-C* alternative dans un navigateur.

dans un des panneaux de la GUI). Ils comprennent notamment les résultats de toutes les analyses effectuées. Par ailleurs, *Frama-C* possède un état global constitué de valeurs mutables bien identifiées, dont une bibliothèque du noyau possède une vue complète [7]. Les résultats fournis par la plateforme ne dépendent que de cet état global et des éventuelles entrées (options de ligne de commande, contenu des fichiers lus, de certaines variables d'environnement, ...). Dès lors, les effets sont les modifications apportées à une des valeurs mutables constituant l'état global et, couplé au fait que *Frama-C* est déterministe, cela garantit *P2* et même plus (notamment, tous les composants de la GUI dépendants de l'état global de *Frama-C* – ce qui inclut toutes les cases à cocher, zones textuelles et autres composants graphiques permettant de paramétrer l'outil – sont automatiquement positionnés dans le bon état après lecture du journal). En outre, on peut considérer que le journal est dynamiquement chargé par *Frama-C* suffisamment tôt pour qu'aucune modification de l'état global n'ait pu survenir (voir [9], section 4.14), ce qui garantit *P3*[2].

2.3. Principe de la journalisation dans *Frama-C*

La génération d'un journal requiert, d'une manière ou d'une autre, d'instrumenter le programme pour *intercepter suffisamment* d'opérations afin de capturer l'ensemble des effets ayant un impact sur les résultats à observer. Ensuite, il faut *modifier* l'exécution de chacune de façon à provoquer aussi son écriture dans le journal. Dans *Frama-C*, il faut donc *intercepter* et *modifier* le comportement de *suffisamment* de fonctions de son API pour capturer toute modification de son état global.

Dès lors, trois questions se posent. D'abord, comment intercepter les opérations ? Ensuite, comment les modifier ? Enfin, comment faire pour qu'il y en ait suffisamment afin de ne rater aucun effet ?

La bibliothèque présentée ici apporte une réponse simple : il est de la responsabilité du développeur de choisir les fonctions à journaliser, c'est-à-dire à écrire dans le journal, *via* un appel à `Journal.register`, qui prend en argument une fonction à journaliser, et l'enregistre comme telle, en effectuant au passage les modifications nécessaires. Celles-ci constituent le cœur de la journalisation et sont décrites en section 4. Nous expliquons aussi en fin de section comment *Frama-C* ne laisse pas, malgré tout, l'entière responsabilité du choix des fonctions à journaliser au développeur. La réponse à la dernière question est plus compliquée, en particulier dans le cas d'un logiciel de la taille de *Frama-C* (près de 150 *kloc* dans sa version libre, avec un état global constitué de près de 1000 valeurs mutables). En considérant le graphe d'appel complet de *Frama-C*, la condition à remplir est la suivante :

> si l'état global de *Frama-C* est modifié pendant l'appel à une fonction f, il doit y avoir au moins une fonction journalisée sur tout chemin entre le point d'entrée de *Frama-C* et f *(P4)*.

Notons que *P4* impose d'avoir *au moins* une fonction journalisée par chemin d'exécution, mais ne requiert pas d'en avoir *exactement* une. Le cas où plusieurs appels de fonctions journalisées sont imbriqués est détaillé en section 4.4.

Deux cas extrêmes peuvent facilement être considérés pour garantir *P4* : soit journaliser le point d'entrée de *Frama-C* exécuté à l'issue de la phase d'initialisation, soit journaliser les fonctions feuilles produisant directement les effets sur l'état global. Malheureusement, aucune de ces deux solutions n'est satisfaisante. Dans le premier cas, le journal ne contiendrait alors qu'un seul appel de fonction, celui du point d'entrée, ce qui rend caduque son utilisation comme support à l'écriture de script par un utilisateur et, surtout, ne permet pas de rendre compte des actions utilisateurs dans la GUI. Dans le

2. En réalité, ce n'est pas totalement exact : certains effets très particuliers ont lieu très tôt lors de la phase d'initialisation, comme par exemple le fait de passer en mode débogage, et peuvent avoir une influence sur cette dernière en terme d'affichages produits, violant ainsi *P3* : même si ces effets sont enregistrés dans le journal, celui-ci n'étant pas encore chargé en mémoire, ces affichages ne peuvent pas avoir lieu lors de son exécution. Néanmoins, aucun de ces affichages n'est critique. En outre, la première action effectuée par le journal sera d'exécuter ces effets, rétablissant ainsi *P3* en prenant comme état initial de P celui obtenu à la fin de la phase d'initialisation et comme état initial de J celui obtenu à la fin de cette action préliminaire.

second cas, il devient nécessaire d'exporter dans l'API de *Frama-C* l'ensemble des fonctions feuilles de bas niveau produisant des effets, ce qui pose un grave problème en terme de préservation d'invariants internes et de maintenabilité de l'outil.

Il convient donc d'apporter une solution pragmatique intermédiaire, qui ne laisse pas la responsabilité de la correction du journal aux seuls contributeurs, notamment extérieurs (dont le nombre croît et est non borné du fait du caractère ouvert de la plateforme). L'architecture de *Frama-C* nous offre ici une solution simple. Comme les greffons doivent enregistrer leur API dans le noyau, il suffit principalement de journaliser ces fonctions ou, en tout cas, celles d'entre elles ayant un effet sur l'état global. En pratique, les fonctions configurant les greffons sont ainsi automatiquement journalisées, tandis que la journalisation des autres consiste juste à positionner un drapeau booléen au moment de l'enregistrement de chacune d'elles dans le noyau. Le choix de journalisation une fonction ou non est en outre facile à effectuer selon qu'elle effectue des calculs utiles à un analyseur ou qu'elle se contente d'exploiter un résultat calculé par ailleurs. Pour être tout à fait complet, il convient aussi de journaliser quelques fonctions du noyau accessibles directement depuis la GUI en dehors de toute analyse, mais leur nombre demeure limité.

2.4. Exemple

Voici à présent un exemple permettant de rendre le journal plus concret. Nous commençons par lancer la GUI de *Frama-C* sur les fichiers `.c` du répertoire courant avec la ligne de commande suivante.

```
$ frama-c-gui -journal-enable *.c
```

L'option `-journal-enable` permet de déclencher la génération du journal. Sans cette option, il n'est généré que lorsque l'utilisation de la GUI provoque une erreur interne. Une fois la GUI lancée, nous exécutons l'analyse de valeurs par interprétation abstraite, puis nous sélectionnons une instruction s et formulons une requête de slicing demandant à générer un programme C préservant la sémantique de s : un nouveau programme C est alors généré dans un nouveau projet *Frama-C*, tandis que les instructions conservées sont mises en évidence à l'écran. Nous nous déplaçons à présent dans le nouveau projet pour inspecter le code généré, puis quittons *Frama-C*. À cet instant, le journal est créé dans le fichier `frama_c_journal.ml` dont la fonction principale est présentée figure 2.

Cette fonction reproduit les effets des actions utilisateurs sur l'état global de *Frama-C* en appelant des fonctions de son API. On peut notamment constater que le code est mis en forme et raisonnablement lisible, que des structures de données complexes peuvent être générées (listes, tables d'association, types de données internes à *Frama-C*), que des variables locales sont créées si nécessaire, puis utilisées et que les étiquettes et les arguments optionnels sont correctement gérés. Pour rejouer la session utilisateur, il suffit maintenant d'exécuter la commande suivante.

```
$ frama-c-gui -load-script frama_c_journal.ml
```

3. Typage dynamique dans *Frama-C*

Le mécanisme de journalisation détaillé dans cet article s'appuie sur la bibliothèque de typage dynamique de *Frama-C*. Nous nous contentons ici d'en rappeler les caractéristiques utiles à notre étude. Le lecteur curieux peut se référer à [8] pour plus de détails, notamment concernant son implantation.

Pour chaque type τ, cette bibliothèque permet de construire une unique *valeur de type* de type τ `Type.t` représentant à l'exécution le type τ. Elle fournit des valeurs de type pour les types primitifs,

3. Ici, les commentaires triplement étoilés *(*** ... ***)* ont été ajoutés manuellement *a posteriori* afin d'expliquer le code généré, mais il pourrait l'être automatiquement. Par ailleurs, le journal complet, incluant le code ne variant pas d'une génération à l'autre, est en annexe A.

```
(* Run the user commands *)
let run () =
  (*** positionnement des fichiers .c à analyser ***)
  Dynamic.Parameter.StringList.set ""
    [ "CruiseControl.c"; "CruiseControl_const.c" ];
  (*** construction de l'AST à partir des fichiers .c donnés ***)
  File.init_from_cmdline ();
  (*** exécution de l'analyse de valeurs ***)
  !Db.Value.compute ();
  (*** construction d'une requête de slicing ***)
  let cil_datatype__Varinfo__t_map =
    !Db.Slicing.Select.select_stmt
      Db.Slicing.Select.empty_selects
      ~spare:false
      (fst (Kernel_function.find_from_sid 306))
      (Globals.Functions.find_by_name "CruiseControl")
  in
  (*** construction d'un projet de slicing ***)
  let sl_project_Slicing1 = !Db.Slicing.Project.mk_project "Slicing1" in
  (*** ajout de la requête précédente au projet de slicing ***)
  !Db.Slicing.Request.add_persistent_selection
    sl_project_Slicing1
    cil_datatype__Varinfo__t_map;
  (*** application de la requête de slicing ***)
  !Db.Slicing.Request.apply_all_internal sl_project_Slicing1;
  (*** suppression des fonctions non appelées dans le code généré ***)
  !Db.Slicing.Slice.remove_uncalled sl_project_Slicing1;
  (*** création du nouveau projet contenant le code généré ***)
  let p_Slicing1__export =
    !Db.Slicing.Project.extract "Slicing1 export" sl_project_Slicing1
  in
  (*** changement de projet courant ***)
  Project.set_current p_Slicing1__export;
  ()
```

FIGURE 2 – Exemple de code généré dans le journal. [3]

tout en permettant au développeur d'en créer de nouvelles. Ainsi, la valeur `Datatype.int`, de type `int Type.t`, représente le type `int`, tandis que la valeur de type `Datatype.func (Datatype.list Datatype.string) Datatype.unit`, de type `(string list → unit) Type.t`, représente le type `string list → unit`. Cette bibliothèque permet de représenter n'importe quel type, y compris les types abstraits en préservant l'abstraction, à l'exception des types polymorphes pour lesquels des générateurs de valeurs de type pour leurs instances monomorphes sont fournis. Les étiquettes et les arguments optionnels sont également pris en charge pour les valeurs de type fonctionnelles.

Grâce à ces valeurs de type, la bibliothèque propose des fonctions génériques bien typées. On trouve ainsi la fonction `Datatype.pretty_code` de type α `Type.t` $\rightarrow$ `Format.formatter` $\rightarrow \alpha \rightarrow$ `unit` qui permet d'afficher une valeur sous la forme d'un code *OCaml* correct. Le type dynamique permet de sélectionner la fonction d'affichage adéquate en fonction de la valeur de type v fournie, le typeur d'*OCaml* garantissant que v coïncide avec le type de l'argument à afficher. Le développeur peut définir cette fonction pour chacun de ses propres types de données. Dans le cas des types abstraits, le

code généré ne doit dépendre que des fonctions visibles dans l'API. Par exemple, la fonction d'affichage des `kernel_function` de *Frama-C*, représentant les fonctions *C* annotées, génère un code appelant `Globals.Functions.find_by_name` qui recherche une `kernel_function` à partir de son nom :

```
let pretty_code fmt kf =
  Format.fprintf
    fmt "Globals.Functions.find_by_name \"%s\"" (Kernel_function.get_name kf)
```

Différentes tables hétérogènes sont également proposées. Par exemple, l'interface des tables hétérogènes sur les clés et polymorphiquement homogènes sur les valeurs associées est la suivante [4].

```
module Obj_tbl: sig
  type α t                                          (* α = type des valeurs associées *)
  val create: unit → α t
  val add: α t → β Type.t → β → α → unit
  val find: α t → β Type.t → β → α
end
```

Ici, la valeur de type v fournie en argument aux fonctions `Obj_tbl.add` et `Obj_tbl.find` permet de garantir statiquement que le type de la clé coïncide avec v au moment de l'application de la fonction.

Par ailleurs, la bibliothèque fournit quelques possibilités supplémentaires. Ainsi, la fonction `Type.Function.get_gadt_instance` de type α `Type.t` $\to \alpha$ `Type.Function.gadt_instance` permet d'obtenir, pour une valeur de type fonctionnelle représentant un type $\tau_1 \to \tau_2$, les valeurs de type correspondant à τ_1 et τ_2, ainsi que les éventuels étiquette et valeur par défaut associés à l'argument. Le type α `Type.Function.gadt_instance` est ici un type algébrique gardé (GADT) [10, 6] de manière à établir un lien entre le type fonctionnel et les types de son argument et de sa valeur de retour : [5]

```
type _ gadt_instance =
  | No_instance
  | Instance: α Type.t × β Type.t → (α → β) gadt_instance
```

Ce GADT permet de garantir statiquement que la fonction `Type.Function.get_gadt_instance` ne peut renvoyer le constructeur `Instance` que si elle est appelée avec, comme argument, une valeur d'un type $(\tau_1 \to \tau_2)$ `Type.t`. Il garantit alors que ses première et deuxième composantes sont respectivement des valeurs de type τ_1 `Type.t` et τ_2 `Type.t`, correspondant aux valeurs de type de l'argument et du résultat, respectivement. Dans ce cas particulier, l'implémentation (non montrée) de la fonction `Type.Function.get_gadt_instance` garantit aussi la réciproque : si elle renvoie `No_instance`, alors son argument n'est pas une valeur de type fonctionnelle.

4. Journaliser des fonctions *OCaml*

La modification des fonctions permettant l'écriture de leur appel dans le journal est effectuée au moment de leur enregistrement dans la bibliothèque de journalisation *via* la fonction `Journal.register`. Cette fonction est un *wrapper* prenant en argument une fonction quelconque et renvoyant une fonction observationnellement équivalente *modulo* l'écriture supplémentaire dans le journal. Idéalement, son type attendu serait donc le suivant.

4. Seules les fonctions principales sont présentées ici. Plus de détails sur l'implantation des tables hétérogènes sont fournis dans l'article associé [8].

5. La version actuelle de *Frama-C* (Fluorine-20130601) doit être compatible avec la version 3.12.1 d'*OCaml*. Elle ne peut donc pas utiliser de GADTs, seulement introduits dans la version 4 du compilateur. Ainsi, elle utilise actuellement une version sans GADT, mais comprenant quelques occurrences de `Obj.magic` pour contourner des problèmes de typage.

`val register:` $(\alpha \to \beta) \to \alpha \to \beta$

Il est en réalité légèrement différent, comme nous allons l'expliquer. Plus généralement, plutôt qu'introduire l'algorithme de journalisation *in extenso*, nous allons partir du cas le plus simple (les fonctions de type `unit → unit`) pour introduire progressivement les cas plus généraux et complexes.

4.1. Fonctions de type `unit → unit`

La première difficulté consiste à imprimer le nom de chaque fonction f à journaliser : ce n'est pas possible en *OCaml*. Pour la contourner, nous demandons au développeur de fournir le nom de f tel qu'il apparaît dans l'API du logiciel. Le type de `Journal.register` devient alors le suivant.

```
val register: string → (unit → unit) → unit → unit
```

Dès lors, coder cette fonction est en fait très simple, en supposant l'existence d'une valeur `fmt` de type `Format.t` correspondant au canal d'écriture considéré (le journal) [6].

```
(* fonction d'écriture dans le journal *)
let print_sentence name fmt = Format.fprintf fmt "%s ();" name

let register name f () =
  (* écriture dans le journal *)
  print_sentence name fmt;
  (* exécution de la fonction journalisée *)
  f ()
```

Prenons l'exemple de la fonction du noyau de *Frama-C* `File.init_from_cmdline: unit → unit`, utilisée dans la figure 2, et qui génère un AST à partir des fichiers *C* fournis préalablement. Journaliser cette fonction revient à exécuter le code suivant.

```
(* file.ml *)
let init_from_cmdline () = ...
let init_from_cmdline = Journal.register "File.init_from_cmdline" init_from_cmdline
```

Le fait de n'appliquer que partiellement la fonction `Journal.register` est fondamental : la fermeture renvoyée correspond à la fonction journalisée modifiée qui, à chaque application, effectue son effet et l'écrit dans le journal.

En pratique, plutôt que d'écrire directement dans le canal `fmt`, nous utilisons une file enregistrant les commandes d'affichage à exécuter, pour ensuite générer le journal à la fin de l'exécution. Le code modifié de `Journal.register` devient alors le suivant.

```
(* générateur d'instructions dans le journal *)
module Sentences: sig
  val add: (Format.formatter → unit) → unit          (* ajout d'une instruction *)
  val write: Format.formatter → unit      (* écriture de toutes les instructions *)
end = struct
  let sentences: (Format.formatter → unit) Queue.t = Queue.create ()
  let add pp = Queue.add pp sentences
  let write fmt = Queue.iter (fun pp → pp fmt) sentences; Format.fprintf fmt "()"
```

6. Par soucis de clarté, ce code ne prend pas en compte le fait que l'option `-journal-enable` ait été ou non positionnée : on suppose que c'est toujours le cas.

```
end

let register name f () =
  Sentences.add (print_sentence name);
  f ()
```

4.2. Arguments d'un type quelconque

Lorsque l'argument de la fonction à journaliser n'est pas `()`, son impression dépend de son type et de sa valeur à l'exécution. Pour effectuer cette impression, on utilise la fonction `Datatype.pretty_code` de la bibliothèque de typage dynamique présentée en section 3 pour afficher des arguments quelconques. Cela requiert néanmoins de prendre un type dynamique en argument.

```
let print_sentence name ty_x x fmt =
  Format.fprintf fmt "%s %a;" name (Datatype.pretty_code ty_x) x

let register name ty_x f x =
  Sentences.add (print_sentence name ty_x x);
  f x
```

Le type de la fonction `Journal.register` est donc le suivant.

```
val register: string → α Type.t → (α → unit) → α → unit
```

Ainsi, la fonction du noyau de *Frama-C* `Project.set_current: Project.t → unit` qui change le projet courant est-elle journalisée de la manière suivante.

```
(* project.ml *)
type t = ...                                                    (* type des projets *)
let ty: t Type.t = ...                                        (* valeur de type des projets *)
let set_current prj = ...
let set_current = Journal.register "Project.set_current" ty set_current
```

Arguments fonctionnels Les valeurs de certains types de données ne peuvent néanmoins pas être affichés sous la forme de code *OCaml* sans indication externe. C'est tout particulièrement le cas des fonctions, pour lesquelles il faudrait être en mesure de générer du code pour les fermetures. Ici, nous utilisons une table hétérogène globale, fournie par la bibliothèque de typage dynamique, de manière à associer à chaque valeur non affichable une chaîne de caractères la représentant : cette dernière est utilisée pour l'écriture dans le journal. Par exemple, pour les fonctions, cette chaîne est celle apparaissant dans l'API du logiciel. À ce stade, cette table ne contient d'ailleurs que des fonctions, mais d'autres valeurs non fonctionnelles y seront ajoutées en section 4.3. Même si elle grossit de manière monotone, sa taille demeure petite en pratique.

```
(* associe une chaîne à une valeur *)
module Binding: sig
  val add: α Type.t  → α → string  → unit
  val find: α Type.t  → α → string
end = struct
  let bindings: string Type.Obj_tbl.t = Type.Obj_tbl.create ()
  let add ty v var = Type.Obj_tbl.add bindings ty v var
  let find ty v = Type.Obj_tbl.find bindings ty v
end
```

Les fonctions journalisées sont automatiquement ajoutées dans cette table. Les autres éventuelles doivent l'être manuellement par le développeur, mais cela demeure exceptionnel dans *Frama-C*[7]. La fonction d'affichage des appels journalisés est maintenant la suivante.

```
let pp ty fmt x =
  try
    let name = Binding.find ty x in
    (* utiliser la chaîne pré-enregistrée si elle existe *)
    Format.fprintf fmt "%s" name
  with Not_found →
    (* utiliser l'afficheur par défaut sinon *)
    Datatype.pretty_code ty fmt x;

(* utiliser pp plutôt que l'afficheur générique *)
let print_sentence name ty_x x fmt = Format.fprintf fmt "%s %a;" name (pp ty_x) x
```

En outre, la fonction d'enregistrement associe aussi le nom à la fermeture *via* le module `Binding`.

```
let register name ty_x f x =
  (* auto-enregistrement de la fonction journalisée *)
  Binding.add (Datatype.func ty_x Datatype.unit) f name;
  Sentences.add (print_sentence name ty_x x);
  f x
```

Polymorphisme Il s'agit de la principale limitation de la journalisation : la bibliothèque de typage dynamique ne supportant pas les types polymorphes (mais uniquement leurs instances monomorphes), il n'est pas possible de journaliser de fonctions polymorphes. Néanmoins, de telles fonctions ne peuvent de toute façon pas être dans les APIs des greffons [8] qui définissent la (quasi) totalité des fonctions à journaliser. En outre, les greffons représentent des analyseurs : ils n'ont pas spécialement vocation à être générique. De ce fait, l'absence de polymorphisme n'est pas un problème pratique dans *Frama-C*.

4.3. Valeurs de retour

Si une fonction journalisée f renvoie une valeur v, il est possible qu'une autre fonction journalisée g la prenne en argument. Dans ce cas, créer une nouvelle valeur v' observationnellement équivalente à v n'est pas nécessairement correct, en particulier si v est représentée à l'exécution par un pointeur. La seule représentation correcte consiste à lier le résultat de f à l'argument de g à l'aide d'une variable locale pour préserver l'égalité physique `==`.

Pour ce faire, nous allons utiliser à nouveau le module `Binding`, introduit à la section 4.2 : lorsque le type de retour de la fonction n'est pas `unit`, nous générons une nouvelle variable locale pouvant être utilisée dans la suite du journal. En supposant l'existence d'un générateur de noms de variables *OCaml* `gen_binding: unit → string`, la fonction `print_sentence` est modifiée comme suit.

```
let print_sentence name ty_x x fmt =
  (* ancienne version de la fonction *)
  let pp_sentence fmt = Format.fprintf fmt "%s %a" name (pp ty_x) x in
  if Type.equal ty_x Datatype.unit then Format.fprintf fmt "%t;" pp_sentence
```

7. Dans la version actuelle de *Frama-C*, on ne trouve qu'un unique ajout non automatique. Il s'agit d'une fonction non journalisée pouvant être donnée en argument d'une journalisée d'ordre supérieur. Le cas est rare car les fonctions journalisées sont les fonctions de haut niveau, qui sont, d'une part, rarement d'ordre supérieur et, d'autre part et de par l'architecture de *Frama-C*, rarement appelées avec pour argument une fonction de (plus) bas-niveau.

```
  else
    (* ajoute une liaison locale *)
    let v = gen_binding () in
    Binding.add ty_x x v;
    Format.fprintf fmt "let %s = %t in " v pp_sentence
```

Le corps de la liaison locale est généré par la suite de l'exécution (dans le cas de la fin d'exécution, un `()` terminal est ajouté par la fonction `Sentences.write` de la section 4.1). Aussi le test effectué ici n'est pas strictement indispensable, mais il le deviendra dans la section 4.6. Il permet en outre de générer un code plus lisible qu'un `let () = ... in ...`.

4.4. Appels de fonctions elles-mêmes journalisées

Dans la section 2.3, nous avons expliqué qu'il fallait journaliser suffisamment de fonctions pour ne rater aucun effet produit par le logiciel. En contrepartie, le risque est d'avoir *trop* d'appels à des fonctions journalisées écrites dans le journal. En effet, considérons le cas d'une fonction f journalisée appelant une autre fonction g elle-même journalisée : lors d'un appel à f avec un argument x, $f\ x$ est écrit dans le journal, puis l'appel est effectivement exécuté engendrant l'appel de la fonction g à un argument y, générant à son tour l'écriture dans le journal de $g\ y$ et l'appel correspondant. *In fine*, le journal généré contient la séquence `f x; g y` et son exécution provoque l'exécution de f, incluant celle de g, puis une nouvelle exécution de g, ce qui n'est pas équivalent au programme initial : la fonction g a été exécutée une fois de trop.

Pour résoudre ce problème, nous suspendons l'écriture dans le journal des fonctions journalisées lorsqu'une est déjà en cours d'exécution, grâce à une référence additionnelle.

```
(* !started vaut true ssi une fonction journalisée est en cours d'appel *)
let started = ref false
let register name ty_x f x =
  Binding.add Datatype.func ty_x Datatype.unit f name;
  if !started then f x
  else begin
    (* n'écrire que s'il n'y a pas déjà d'écritures en cours *)
    Sentences.add (print_sentence name ty_x x);
    (* maintenant, la journalisation est en cours *)
    started := true;
    f x;
    (* restauration de la valeur initiale *)
    started := false
  end
```

Notons que l'utilisation de cette référence ne permet pas d'utiliser le journal dans un contexte concurrent. Néanmoins, nous nous sommes déjà placés dans un cadre déterministe (cf. section 2.1).

4.5. Exceptions

Une des premières utilisations du journal est de reproduire les erreurs survenues au cours de l'exécution. En *OCaml*, ces erreurs correspondent à des levées d'exceptions. La journalisation doit les prendre doublement en compte : d'une part, la levée d'une exception ne doit pas entraver l'écriture dans le journal et, d'autre part, le code généré doit prendre en compte proprement cette exception dont on sait à l'avance qu'elle surviendra lors de son exécution.

Pour continuer à écrire dans le journal en présence de comportements exceptionnels, il faut aussi effectuer le post-traitement dans ces cas-là. Aussi, pour prendre en compte les exceptions, lorsqu'une est levée, nous devons la rattraper dans le code produit de manière à générer la suite de l'exécution dans le cas exceptionnel, tandis que la suite du cas nominal n'est jamais exécutée : on peut donc y générer un `assert false`. Le code écrit dans le journal est donc différent selon que la fonction journalisée lève ou non une exception. Le code est donc maintenant généré *après* l'exécution de la fonction, selon deux schémas différents, et non plus avant. On obtient donc le code suivant.

```
let register name ty_x f x =
  Binding.add Datatype.func ty_x Datatype.unit f name;
  if !started then f x
  else begin
    started := true;
    try
      f x;
      (* cas nominal: aucune exception levée, comme précédemment *)
      Sentences.add (print_sentence name ty_x x);
      started := false
     with exn →
       (* cas exceptionnel: gérer l'exception dans le journal *)
       Sentences.add
         (fun fmt →
            Format.fprintf fmt "try %t assert false with exn (* %s *) → "
              (print_sentence name ty_x x)
              (Printexc.to_string exn));
       started := false;
       (* re-lever l'exception pour préserver le comportement de la fonction *)
       raise exn
  end
```

À présent, les exceptions sont gérées correctement par le journal. Par exemple, considérons le schéma de code suivant dans lequel les fonctions `f` et `g` sont journalisées mais par `h`.

```
let f () = ... raise E ...
let g () = ...
let h () = try ... f () ... with E → g ()
```

Si l'exception `E` est effectivement levée lors de l'appel à `f` effectué *via* `h`, alors le journal généré contiendra le code suivant.

```
let run () =
  ...
  try f (); assert false with exn (* E *) → g (); ...
```

Néanmoins, si l'exception `E` n'était jamais rattrapée (et notamment pas dans `h`), le logiciel serait arrêtée sur la levée de cette exception et l'exécution du journal reproduirait ce comportement. C'est le comportement initialement souhaité : le journal a le même comportement que le programme dont il est issu. Cependant, d'une part il devient difficile pour l'utilisateur de distinguer le cas où le journal reproduit l'exception initiale et le cas éventuel où l'exécution du journal lève une exception car un problème quelconque est survenu (même si ce n'est pas censé survenir) et, d'autre part, arrêter l'exécution du logiciel sur une exception alors qu'on a obtenu le comportement attendu (le journal a reproduit le comportement initial) n'est pas souhaitable.

Pour cette raison, si le programme initial lève une exception `E` jamais rattrapée, le code généré dans le journal la lève à son tour, mais encapsulée dans une autre – l'exception `Exception E` – de façon à pouvoir la distinguer des autres (par exemple, celles levées par une exécution du journal qui ferait échouer le logiciel pour une raison inconnue). Ainsi, la fonction principale du journal [8], qui appelle la fonction `run`, peut rattraper cette exception particulière pour afficher un message à l'utilisateur (indiquant que le programme initial avait levé telle exception), avant d'arrêter proprement le logiciel. Pour générer la levée de cette exception spéciale, le module `Sentences` est modifié de la façon suivante.

```
module Sentences: sig
  val add: (Format.formatter → unit) → bool (* exception levée ? *) → unit
  val write: Format.formatter → unit
end = struct
  type t = { pp: Format.formatter → unit; raise_exn: bool }
  let sentences: t Queue.t = Queue.create ()
  let add pp exn = Queue.add pp { pp = pp; raise_exn = exn } sentences
  let write fmt =
    (* la dernière instruction a levé une exception ? *)
    let finally_raise = ref false in
    Queue.iter (fun s → s.pp fmt; finally_raise := s.raise_exn) sentences;
    (* si le dernier appel journalisé a levé une exception non rattrapée,
       l'encapsuler dans l'exception propre au journal. *)
    Format.fprintf fmt "%s"
      (if !finally_raise then "raise (Exception (Printexc.to_string exn))"
       else "()")
end
```

4.6. Arguments multiples

En *OCaml*, les fonctions à plusieurs arguments de type $\tau_1 \to \tau_2 \to \ldots \to \tau_n$ sont isomorphes aux fonctions prenant un unique argument de type τ_1 et renvoyant une fermeture de type $\tau_2 \to \ldots \to \tau_n$. L'algorithme précédent pourrait donc fonctionner. Néanmoins, générer une nouvelle liaison locale pour chaque application partielle rend le journal illisible. Nous préférons donc modifier l'algorithme pour retarder l'écriture dans le journal d'un appel de fonction jusqu'à son application totale. Ceci suppose néanmoins que les différentes applications partielles n'engendrent pas d'effets de bord, ce qui est le cas en pratique dans *Frama-C*. Ici, le principe est d'itérer récursivement sur le type de la fonction pour construire dans une continuation l'instruction finale à générer, en y ajoutant au fur et à mesure les arguments appliqués. Étendre la continuation de type `Format.formatter → unit` est simple.

```
(* écrit un argument supplémentaire arg dans la continuation f_acc *)
let extend_continuation f_acc pp_arg arg fmt =
  Format.fprintf fmt "%t %a" f_acc pp_arg arg
```

Le cas de base de la récurrence correspond à journaliser un type non fonctionnel, auquel cas la continuation est directement appliquée. Pour ce faire, on modifie la fonction `print_sentence` :

```
let print_sentence f_acc ty fmt =
  if Type.equal ty Datatype.unit then f_acc fmt;
  else
    let v = gen_binding () in
    Binding.add ty x v;
    Format.fprintf fmt "let %s = %t in " v f_acc;
```

8. Présentée en annexe A.

Dans le cas récursif, dans lequel la valeur à journaliser est une fermeture f, il nous faut avoir accès à son argument. Pour cela, nous déconstruisons f, effectuons le traitement attendu (appliquer la fonction à son argument, étendre la continuation et journaliser le résultat), et reconstruisons finalement la nouvelle fermeture. L'algorithme devient alors le suivant (la gestion des exceptions est ici omise).

```
let rec journalize: τ. (Format.formatter → unit) → τ Type.t → τ → τ =
  fun (type t) f_acc (ty: t Type.t) (x:t) →
    match Type.Function.gadt_instance ty with
    | Type.Function.No_instance →
      (* x est une valeur non fonctionnelle:
         l'application (éventuellement 0-aire) en cours est totalement effectuée *)
      (* écriture de la continuation dans le journal *)
      if not !started then Sentences.add (print_sentence f_acc ty);
      (* renvoie du résultat *)
      x
    | Type.Function.Instance(ty_y, ty_res) →
      (* x est une fonction: on reconstruit la fermeture *)
      fun y →
        if !started then x y
        else begin
          started := true;
          (* application de la fonction à son argument *)
          let res = x y in
          started := false
          (* extension de la continuation *)
          let f_acc = extend_continuation f_acc (pp ty_y) y in
          (* journalisation du résultat de l'appel *)
          journalize f_acc ty_res res
        end
```

On peut noter que la fonction ci-dessus est récursivement polymorphe, ce qui explique son annotation de type explicite. Désormais, `Journal.register` se contente d'appeler cette fonction avec la bonne continuation initiale affichant la valeur x à journaliser (*i.e.* le nom de la fonction si x en est une).

```
let register s ty x =
  Binding.add ty x s;
  (* initialisation de la continuation *)
  let f_acc fmt = pp ty fmt x in
  journalize f_acc ty x
```

Contrairement à précédemment, cet enregistrement ne fonctionne plus uniquement sur des fermetures. Son type peut donc être généralisé.

```
val register: name → α Type.t → α → α
```

5. Conclusion

Cet article a présenté le mécanisme de journalisation de *Frama-C*. Il permet de générer automatiquement un script *OCaml* permettant de reproduire automatiquement les actions utilisateurs, notamment effectuées dans la GUI. Le surcoût engendré, aussi bien en temps qu'en mémoire, est totalement négligeable dans le contexte de *Frama-C*.

En pratique, l'algorithme de journalisation possède quelques fonctionnalités qui n'ont pas été présentées dans cet article. Ainsi, les étiquettes et les arguments optionnels sont correctement gérés et affichés le cas échéant dans le code généré. Un effort particulier a été effectué pour générer du code lisible, correctement indenté et minimalement parenthésé. Le code généré peut aussi contenir des commentaires explicatifs.

Par ailleurs, pour être bien typé, cet algorithme combine typage statique et dynamique, *via* un type fantôme, et utilise, certes à faibles doses, plusieurs fonctionalités récemment introduites dans *OCaml* : types localement abstraits, récursion polymorphe et GADTs. En outre, même si elle n'avait jamais été présentée jusqu'à aujourd'hui, la journalisation est présente dans *Frama-C* depuis 2008. L'algorithme n'a pas été modifié depuis, sauf pour intégrer les facilités de typage offertes depuis *OCaml* 3.12. Plus que tout autre argument déjà avancé, cette stabilité est un gage de confiance envers sa correction et envers le fait qu'elle remplit la fonctionnalité souhaitée.

Remerciements Je tiens à remercier chaleureusement François Bobot de m'avoir soufflé l'idée des GADTs et de m'avoir fait part de ses remarques éclairées. Je remercie également Zaynah Dargaye et les rapporteurs anonymes pour leurs commentaires avisés m'ayant permis d'améliorer cet article. Par ailleurs, si Benjamin Monate ne m'avait pas, en 2008, expliqué la journalisation de *Caveat* et soumis le challenge d'implémenter une fonctionnalité similaire en *OCaml* pour *Frama-C*, cet article et l'implémentation qui s'y rapporte n'auraient sans doute jamais été écrits. Enfin, ces travaux ont été soutenus par le projet européen FP7 Stance.

Références

[1] P. Baudin, J.-C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language, v1.7*, April 2013. `http://frama-c.com/acsl.html`.

[2] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. Caveat : a tool for software validation. In *International Conference on Dependable Systems and Networks (DSN'02)*, pages 537+, 2002.

[3] L. Correnson and J. Signoles. Combining Analyses for C Program Verification. In M. Stoelinga and R. Pinger, editors, *Formal Methods for Industrial Case Studies (FMICS'12)*, volume 7437 of *Lecture Notes in Computer Science*, pages 108–130. Springer, August 2012.

[4] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a software analysis perspective. In *International Conference on Software Engineering and Formal Methods (SEFM'12)*, pages 233–247. Springer, October 2012.

[5] P. Cuoq and J. Signoles. Experience report : Ocaml for an industrial-strength static analysis framework. In G. Hutton and A. P. Tolmach, editors, *International Conference of Functional Programming (ICFP'09)*, pages 281–286. ACM, September 2009.

[6] J. Garrigue and J. Le Normand. Adding GADTs to OCaml : a direct approach. In *Workshop on ML (ML'11)*. ACM, September 2011.

[7] J. Signoles. Foncteurs impératifs et composés : la notion de projet dans Frama-C. In A. Schmitt, editor, *Journées Francophones des Langages Applicatifs (JFLA'09)*, volume 7.2 of *Studia Informatica Universalis*, pages 245–280, September 2009. In French.

[8] J. Signoles. Une bibliothèque de typage dynamique en OCaml. In *Journées Francophones des Langages Applicatifs (JFLA'11)*, Studia Informatica Universalis, 2011.

[9] J. Signoles, L. Correnson, and V. Prevosto. *Frama-C Plug-in Development Guide*, April 2013.

[10] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In A. Aiken and G. Morrisett, editors, *Symposium on Principles of Programming Languages (POPL'03)*, pages 224–235. ACM, January 2003.

A. Code complet d'un journal

Cette annexe présente le contenu du fichier journal `frama_c_journal.ml` généré par *Frama-C* (version `Fluorine-20130601`) pour l'exemple de la section 2.4. Le code de la figure 2 est la fonction `run` de ce fichier.

```
(* Frama-C journal generated at 10:53 the 04/10/2013 *)

exception Unreachable
exception Exception of string

(* Run the user commands *)
let run () =
  Dynamic.Parameter.StringList.set ""
    [ "CruiseControl.c"; "CruiseControl_const.c" ];
  File.init_from_cmdline ();
  !Db.Value.compute ();
  let cil_datatype__Varinfo__t_map =
    !Db.Slicing.Select.select_stmt
      Db.Slicing.Select.empty_selects
      ~spare:false
      (fst (Kernel_function.find_from_sid 306))
      (Globals.Functions.find_by_name "CruiseControl")
  in
  let sl_project_Slicing1 = !Db.Slicing.Project.mk_project "Slicing1" in
  !Db.Slicing.Request.add_persistent_selection
    sl_project_Slicing1
    cil_datatype__Varinfo__t_map;
  !Db.Slicing.Request.apply_all_internal sl_project_Slicing1;
  !Db.Slicing.Slice.remove_uncalled sl_project_Slicing1;
  let p_Slicing1__export =
    !Db.Slicing.Project.extract "Slicing1 export" sl_project_Slicing1
  in
  Project.set_current p_Slicing1__export;
  ()

(* Main *)
let main () =
  Journal.keep_file "frama_c_journal.ml";
  try run ()
  with
  | Unreachable → Kernel.fatal "Journal reaches an assumed dead code"
  | Exception s → Kernel.log "Journal re-raised the exception %S" s
  | exn →
    Kernel.fatal
      "Journal raised an unexpected exception: %s"
      (Printexc.to_string exn)

(* Registering *)
let main : unit → unit =
  Dynamic.register
```

```
    ~plugin:"Frama_c_journal"
    "main"
    (Datatype.func Datatype.unit Datatype.unit)
    ~journalize:false
    main

(* Hooking *)
let () = Cmdline.run_after_loading_stage main; Cmdline.is_going_to_load ()
```

De la KAM avec un Processus d'Ordre Supérieur

D. Pous[1] & A. Schmitt[2]

1 : CNRS, damien.pous@ens-lyon.fr
2 : Inria, alan.schmitt@inria.fr

Résumé

Nous présentons un encodage simple et direct de la machine abstraite de Krivine (KAM) dans le calcul de processus d'ordre supérieur HOcore, en utilisant un nombre très restreint de canaux de communication. Cet encodage montre qu'il est possible de capturer l'expressivité du λ-calcul en HOcore dès que l'on fixe l'ordre d'évaluation. Nous donnons également une nouvelle borne inférieure pour le nombre minimal de restrictions nécessaire pour rendre l'équivalence de programmes dans HOcore indécidable. [1]

1. Introduction

Le calcul de processus HOcore est remarquable par sa similarité au λ-calcul, auquel il ajoute une notion de concurrence. Malgré cette similarité, aucun encodage du λ-calcul en HOcore n'a été présenté à ce jour. En effet, l'appariement entre une fonction et son argument est très syntaxique et très rigide en λ-calcul, alors qu'il est beaucoup plus lâche en HOcore car il correspond à la présence simultanée sur le même nom de canal d'un message et d'un récepteur pour ce message. Cette différence cruciale, couplée à l'impossibilité de générer de nouveaux noms de canaux, implique que toute traduction doit fixer le nombre de redex pouvant être activés en parallèle. Ce nombre pouvant être non-borné pour certains λ-termes, ceci empêche toute traduction *tant que la stratégie d'évaluation n'a pas été fixée.* C'est cette deuxième voie que nous explorons ici, en choisissant une stratégie en appel par nom, déclinée sous la forme d'une machine abstraite de Krivine (KAM).

Une fois ce choix de conception arrêté, la traduction est très naturelle : on représente la structure récursive de la pile de la KAM comme deux messages transportant respectivement le terme de tête et, récursivement, la queue de la pile. La β-réduction est simplement la communication entre le terme actif, représentant la fonction, avec la tête de la pile. De manière surprenante, cette traduction permet également de capturer l'opérateur de contrôle call-cc de la KAM. La réification de la continuation en tant que pile contenue dans un message permet en effet de facilement la dupliquer ou la remplacer.

Les leçons que l'on peut tirer de cet encodage sont doubles. Tout d'abord, en ce qui concerne l'expressivité de HOcore, il montre comment on peut s'appuyer sur l'ordre supérieur pour atteindre un calcul Turing complet. Les travaux précédents [5] étudiaient l'expressivité en se basant sur des machines de Minsky, qui n'ont besoin que de savoir compter et de détecter qu'un compteur vaut 0. L'ordre supérieur n'est pas nécessaire pour compter, il est en revanche utilisé pour détecter l'égalité à 0 (voir [3] et [1] pour des versions de calculs sans ordre supérieur utilisant d'autres fonctionnalités pour détecter le 0). La seconde leçon porte sur la décidabilité de la congruence barbue. En effet, nous avons montré précédemment que la congruence barbue est décidable pour HOcore [5, 2] si aucun nom de canal n'est caché, et cette congruence devient indécidable si quatre noms de canaux sont cachés. Notre traduction n'ayant besoin que de deux noms de canaux, il permet d'affiner le nombre minimal

1. Ce travail a bénéficié du soutien de l'Agence Nationale pour la Recherche dans le cadre du projet PiCoq ANR 10 BLAN 0305.

de restrictions globales nécessaire pour rendre la congruence barbue forte indécidable : il en suffit de deux.

Le reste de ce papier est organisé comme suit. Nous présentons le calcul HOcore en section 2, et la KAM en section 3. La traduction et sa preuve de correspondance opérationnelle est présentée en section 4 avant de conclure en section 5.

2. Présentation de HOcore

2.1. Syntaxe

Le calcul HOcore [5] peut être vu comme une restriction du π-calcul d'ordre supérieur [7], auquel on aurait enlevé l'opérateur de restriction de nom. Il peut également être vu comme un λ-calcul parallèle, où l'application d'une fonction $\lambda x.Q$ à un argument P est remplacée par une communication sur un canal a entre un envoi de message $\overline{a}\langle P\rangle$ mis en parallèle d'une réception de message $a(x).Q$.

La syntaxe de HOcore est la suivante.

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

Un processus P peut soit être une réception de message sur un certain canal, notée $a(x).P$, soit une émission de message, notée $\overline{a}\langle P\rangle$, soit la mise en parallèle de processus $P \parallel Q$, soit une variable x, soit le processus inactif $\mathbf{0}$. Nous distinguons les *variables* x, y, z des *noms de canaux* a, b, c.

Intuitivement, l'unique règle de réduction est la règle de communication suivante, où l'opération $[P \,/\, x]Q$ est la *substitution* de la variable x par le processus P dans Q. Le processus P, émis sur le canal a, est transmis au préfixe de réception $a(x).Q$.

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

Notons que le calcul est asynchrone : l'émission de message n'a pas de continuation. Dans un calcul synchrone, l'envoi de message est de la forme $\overline{a}\langle P\rangle.Q$, où le processus Q est la continuation démarrée après l'émission du message sur a. Nous montrerons dans la section 4.3 que les messages synchrones permettent un encodage de la KAM ne nécessitant qu'un seul nom de canal.

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

La machine de Krivine pour le λ-calcul en appel par nom [4] permet de ne considérer que des termes clos, et d'éviter les difficultés usuelles dues à la capture de variables libres. Il en est de même avec l'encodage que nous présentons ici : nous ne manipulerons que des termes dont toutes les variables seront liées. Les noms de canaux sont eux tous libres, HOcore n'ayant pas d'opérateur de restriction.

2.2. Sémantique

La sémantique de HOcore est définie par un système de transitions étiquetées (LTS). Les étiquettes sont définies par la syntaxe suivante :

$$\alpha ::= \overline{a}\langle P\rangle \mid a(P) \mid \tau \;\; .$$

Elles correspondent respectivement à l'émission d'un processus, la réception d'un processus, et à une communication interne. Le LTS est défini inductivement, par les règles suivantes.

$$\frac{}{\overline{a}\langle P\rangle \xrightarrow{\overline{a}\langle P\rangle} \mathbf{0}}\ \textsc{Out} \qquad \frac{}{a(x).Q \xrightarrow{a(P)} [P\,/\,x]Q}\ \textsc{Inp}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q}\ \textsc{Par1} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'}\ \textsc{Par2}$$

$$\frac{P \xrightarrow{\overline{a}\langle R\rangle} P' \qquad Q \xrightarrow{a(R)} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}\ \textsc{Tau1} \qquad \frac{P \xrightarrow{a(R)} P' \qquad Q \xrightarrow{\overline{a}\langle R\rangle} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}\ \textsc{Tau2}$$

Congruence structurelle Notons que le LTS précédent est très rigide : il conserve strictement la structure des termes (à l'inverse, une sémantique réductionelle utilise généralement une notion de congruence structurelle, qui permet par exemple de réorganiser les parenthèses modulo associativité de la mise en parallèle). L'avantage est que les termes obtenus après une réduction sont plus facile à analyser ; l'inconvénient est que ce LTS tend à générer de nombreuses occurrences du processus **0**, qui sont inutiles. Par exemple, on a formellement les transitions suivantes :

$$(\overline{a}\langle P\rangle \parallel \overline{b}\langle Q\rangle) \parallel a(x).b(y).(x \parallel y) \xrightarrow{\tau} (\mathbf{0} \parallel \overline{b}\langle Q\rangle) \parallel b(y).(P \parallel y) \xrightarrow{\tau} (\mathbf{0} \parallel \mathbf{0}) \parallel (P \parallel Q)$$

On va donc s'appuyer sur une notion de congruence structurelle très restreinte qui permet de s'affranchir de ces occurrences de **0**.

Une relation $\mathcal{R}$ est une *congruence* si c'est une relation d'équivalence (réflexive, symétrique, transitive) qui respecte les différents constructeurs du langage (par exemple si $P\ \mathcal{R}\ Q$ alors nous avons $a(x).P\ \mathcal{R}\ a(x).Q$). On quotiente dans la suite l'ensemble des processus par la plus petite congruence telle que la composition parallèle admette le processus **0** comme élément neutre à gauche et à droite.

2.3. Expressivité

HOcore est qualifié de minimal, car il ne contient que le strict nécessaire à l'ordre supérieur. Par exemple, il n'inclut pas d'opérateurs de restriction ou de réplication. HOcore est tout de même Turing complet : un encodage fidèle des machines de Minsky est présenté dans [5]. En particulier, le problème de la terminaison y est indécidable.

Un objectif de ce papier est de montrer qu'il est possible d'encoder directement des modèles de calcul plus complexes, comme le λ-calcul, dans HOcore.

2.4. Équivalence de processus

Une des questions cruciales de l'étude de calculs de processus est de savoir si deux processus « font la même chose ». Ainsi, dans l'optique de la programmation modulaire, on doit être capable de dire si deux bibliothèques logicielles sont interchangeables. Deux processus sont équivalents si, dans n'importe quel contexte, ce que l'on observe de leur activité est similaire. Formellement, on définit la *congruence barbue* [6] comme la plus grande relation symétrique telle que :

- si $P \simeq Q$ et $P \xrightarrow{\tau} P'$, alors il existe un Q' tel que $Q \xrightarrow{\tau} Q'$ et $P' \simeq Q'$: la congruence barbue est préservée par réductions ;
- si $P \simeq Q$, alors $C[P] \simeq C[Q]$ pour tout contexte C, un contexte étant un processus avec un trou : la congruence barbue est une congruence ;

– si $P \simeq Q$ et $P \xrightarrow{\overline{a}\langle P''\rangle} P'$, alors il existe Q' et Q'' tels que $Q \xrightarrow{\overline{a}\langle Q''\rangle} Q'$: la congruence barbue met en relation des processus avec les mêmes *observables*, ou barbes, qui sont ici la possibilité d'émettre un message sur un canal donné.

On peut définir de façon similaire la *congruence barbue faible*, notée $\approxeq$, en considérant dans la premiere clause des séquences arbitraires de transitions, plutôt que des transitions simples.

Un des résultats fondamentaux de HOcore est la décidabilité de la congruence barbue [5]. Les processus de HOcore ne pouvant pas cacher leurs réductions (le langage n'a pas d'opérateur de restriction), il est toujours possible de définir des contextes explorant la structure d'un processus. En revanche, dès que l'on autorise des réductions anonymes, par exemple grâce à des restrictions globales empêchant l'observation sur certains noms, la congruence barbue devient indécidable. Les travaux précédents ont montré que quatre telles restrictions globales suffisent.

3. La KAM

La machine abstraite de Krivine est une machine très simple pour évaluer les termes du λ-calcul en appel par nom. Elle permet également de définir des opérateurs de contrôle.

Une configuration de la KAM est composé d'un terme du λ-calcul et d'une pile, qui est une liste de λ-termes. Comme indiqué plus haut, tous les λ-termes considérés sont *clos* (ils ne contiennent pas de variable libre).

$$C ::= M \star \pi$$
$$M ::= x \mid MN \mid \lambda x.M$$
$$\pi ::= M :: \pi \mid []$$

Les règles de réductions de la KAM (sans opérateur de contrôle) sont les suivantes ; un calcul s'arrête quand un état $\lambda x.M \star []$ est atteint.

$$MN \star \pi \mapsto M \star N :: \pi \qquad \text{(Push)}$$
$$\lambda x.M \star N :: \pi \mapsto [N \,/\, x]M \star \pi \qquad \text{(Grab)}$$

La KAM avec opérateurs de contrôles ajoute deux constructions syntaxiques : l'opérateur de capture de continuation `cc`, utilisable dans les programmes, et les constantes de pile k_π, qui ne peuvent être créées que par appel à `cc`. Les deux règles suivantes sont alors ajoutées.

$$\texttt{cc} \star M :: \pi \mapsto M \star k_\pi :: \pi \qquad \text{(CallCC)}$$
$$k_\pi \star M :: \pi' \mapsto M \star \pi \qquad \text{(Restore)}$$

4. KAM en HOcore

Nous commençons par traduire la KAM sans opérateurs de contrôle ; nous étendons ensuite notre encodage à ces opérateurs en section 4.2.

4.1. Version asynchrone

Nous présentons un codage n'utilisant que deux noms libres. La pile courante est un message sur le nom c (“c” pour *continuation*). Le contenu de la pile est un message sur le nom a (“a” comme

argument), correspondant à la tête de la pile, et un message sur c contenant la queue de la pile. La traduction de la pile vide est arbitrairement fixée au processus $\overline{b}\langle\mathbf{0}\rangle$, pour un troisième nom libre b permettant d'observer la fin du calcul (les résultats démontrés ci-dessous sont toujours valides lorsque ce processus est remplacé par un processus arbitraire, tant que celui-ci n'introduit pas de divergence ou de non-déterminisme).

La traduction d'une pile $1 :: 2 :: 3 :: []$ prend donc la forme $\overline{a}\langle 1\rangle \parallel \overline{c}\langle\overline{a}\langle 2\rangle \parallel \overline{c}\langle\overline{a}\langle 3\rangle \parallel \overline{c}\langle\overline{b}\langle\mathbf{0}\rangle\rangle\rangle\rangle$.

Pour ne pas alourdir les notations, nous utilisons le même symbole pour traduire des états, des piles et des λ-termes. Nous supposons également que les noms des variables liées u et s (u n'apparaît que pour la traduction des opérateurs de contrôle) sont différents des noms de variables du λ-terme.

$$
\begin{aligned}
[\![[]]\!] &\triangleq \overline{b}\langle\mathbf{0}\rangle \\
[\![M :: \pi]\!] &\triangleq \overline{a}\langle[\![M]\!]\rangle \parallel \overline{c}\langle[\![\pi]\!]\rangle \\
[\![MN]\!] &\triangleq c(s).([\![M]\!] \parallel \overline{c}\langle\overline{a}\langle[\![N]\!]\rangle \parallel \overline{c}\langle s\rangle\rangle) \\
[\![\lambda x.M]\!] &\triangleq c(s).(a(x).\,[\![M]\!] \parallel s) \\
[\![x]\!] &\triangleq x \\
[\![M \star \pi]\!] &\triangleq [\![M]\!] \parallel \overline{c}\langle[\![\pi]\!]\rangle
\end{aligned}
$$

Notons que dans la traduction d'une configuration, la pile, composée de messages imbriqués sur a et c, est elle même encapsulée à l'intérieur d'un message sur le nom c.

Nous montrons ci-dessous que la réduction de $[\![M \star \pi]\!]$ est déterministe pour tout M et tout π. De plus, à chaque étape de calcul de la KAM correspond une ou deux étapes de calcul du processus traduit (ou trois lorsque l'on prend en compte les opérateurs de contrôle).

Lemme 1 (Substitution)**.** *Pour tous M, x, N avec N clos, nous avons $[\![[N \,/\, x]M]\!] = [[\![N]\!] \,/\, x]\,[\![M]\!]$.*

Démonstration. Par une simple induction sur M. □

Lemme 2 (Simulation)**.** *Pour tous M, M', π, π' tels que $M \star \pi \mapsto M' \star \pi'$, nous avons $[\![M \star \pi]\!] \xrightarrow{\tau}{}^{+} [\![M' \star \pi']\!]$.*

Démonstration. Il suffit de considérer les règles de réduction de la machine de Krivine :

Push ($MN \star \pi \mapsto M \star N :: \pi$) : on vérifie que

$$
\begin{aligned}
[\![MN \star \pi]\!] &= (c(s).\,[\![M]\!] \parallel \overline{c}\langle\overline{a}\langle[\![N]\!]\rangle \parallel \overline{c}\langle s\rangle\rangle) \parallel \overline{c}\langle[\![\pi]\!]\rangle \\
&\xrightarrow{\tau} [\![M]\!] \parallel \overline{c}\langle\overline{a}\langle[\![N]\!]\rangle \parallel \overline{c}\langle[\![\pi]\!]\rangle\rangle \\
&= [\![M \star N :: \pi]\!] \quad .
\end{aligned}
$$

Il suffit donc d'une seule transition pour simuler cette règle.

Grab ($\lambda x.M \star N :: \pi \mapsto [N \,/\, x]M \star \pi$) : il faut cette fois deux transitions :

$$
\begin{aligned}
[\![\lambda x.M \star N :: \pi]\!] &= (c(s).(a(x).\,[\![M]\!]) \parallel s) \parallel \overline{c}\langle\overline{a}\langle[\![N]\!]\rangle \parallel \overline{c}\langle[\![\pi]\!]\rangle\rangle \\
&\xrightarrow{\tau} (a(x).\,[\![M]\!]) \parallel \overline{a}\langle[\![N]\!]\rangle \parallel \overline{c}\langle[\![\pi]\!]\rangle \\
&\xrightarrow{\tau} [[\![N]\!] \,/\, x]\,[\![M]\!] \parallel \overline{c}\langle[\![\pi]\!]\rangle \\
&= [\![[N \,/\, x]M]\!] \parallel \overline{c}\langle[\![\pi]\!]\rangle \qquad \text{(par le Lemme 1)} \\
&= [\![[N \,/\, x]M \star \pi]\!] \quad .
\end{aligned}
$$

□

Il nous faut maintenant prouver que les transitions des processus traduits sont déterministes, et qu'elles correspondent à des réductions dans la KAM. Or, comme on le voit dans la preuve précédente, certaines transitions sont intermédiaires et doivent être complétées afin de correspondre précisément à une réduction de la KAM.

Afin de simplifier la preuve, nous passons par une machine abstraite légèrement différente de la KAM, dans laquelle certaines étapes sont artificiellement dédoublées : les étapes de calcul des processus traduits correspondent ainsi exactement aux réductions de la KAM modifiée.

La modification est la suivante : on introduit une configuration intermédiaire, notée $\lambda' x.M \star \pi$, et la règle Grab est dédoublée comme suit :

$$\lambda x.M \star \pi \mapsto \lambda' x.M \star \pi \qquad (\textsc{Grab}_1)$$
$$\lambda' x.M \star N :: \pi \mapsto [N \,/\, x]M \star \pi \qquad (\textsc{Grab}_2)$$

Fait 3. *Une configuration admet une séquence infinie de réductions dans la KAM originelle si et seulement elle admet une séquence infinie de réductions dans la KAM modifiée.*

En accord avec le second cas dans la preuve du lemme 2, la fonction de traduction est étendue aux configurations intermédiaires en posant :

$$\left[\!\left[\lambda' x.M \star \pi\right]\!\right] \stackrel{\triangle}{=} (a(x).\,[\![M]\!]) \parallel [\![\pi]\!] \quad .$$

Lemme 4. *La fonction de traduction est injective.*

Démonstration. On prouve qu'elle est injective sur les λ-termes, puis sur les piles, puis sur les configurations. □

Notons $[\![\cdot]\!]^{-1}$ la fonction partielle inverse de la traduction, i.e., telle que $[\![[\![C]\!]]\!]^{-1} = C$ pour toute configuration C de la KAM modifiée. On se convainc aisément que cette fonction est calculable.

Lemme 5 (Réflection)**.** *Pour toute configuration C et processus P, si $[\![C]\!] \xrightarrow{\tau} P$, alors $[\![P]\!]^{-1}$ est défini et $C \mapsto [\![P]\!]^{-1}$.*

Démonstration. On raisonne par cas sur la configuration C :
- $C = MN \star \pi$: on a $[\![C]\!] = (c(s).\,[\![M]\!] \parallel \overline{c}\langle \overline{a}\langle [\![N]\!]\rangle \parallel \overline{c}\langle s\rangle\rangle) \parallel \overline{c}\langle [\![\pi]\!]\rangle$, d'où $P = [\![M]\!] \parallel \overline{c}\langle \overline{a}\langle [\![N]\!]\rangle \parallel \overline{c}\langle [\![\pi]\!]\rangle\rangle$ puisqu'une seule transition est possible. On vérifie alors que $[\![P]\!]^{-1} = M \star N :: \pi$, et $C \mapsto [\![P]\!]^{-1}$ par la règle (Push).
- $C = \lambda x.MN \star \pi$: on a $[\![C]\!] = (c(s).(a(x).\,[\![M]\!]) \parallel s) \parallel \overline{c}\langle [\![\pi]\!]\rangle$, d'où $P = (a(x).\,[\![M]\!]) \parallel [\![\pi]\!]$. On vérifie alors que $[\![P]\!]^{-1} = \lambda' x.M \star \pi$, et $C \mapsto [\![P]\!]^{-1}$ par la règle ($\textsc{Grab}_1$).
- $C = \lambda' x.MN \star \pi$: si la pile π est vide, alors $[\![C]\!] = (a(x).\,[\![M]\!]) \parallel \overline{b}\langle \mathbf{0}\rangle$ n'admet pas de transition τ, ce qui contredit l'hypothèse sur P. On a donc $\pi = N :: \pi'$, et $[\![C]\!] = (a(x).\,[\![M]\!]) \parallel \overline{a}\langle N\rangle \parallel \overline{c}\langle [\![\pi']\!]\rangle$. On a nécessairement $P = [[\![N]\!] \,/\, x]\,[\![M]\!] \parallel \overline{c}\langle [\![\pi']\!]\rangle$; on vérifie alors que $[\![P]\!]^{-1} = [N \,/\, x]M \star \pi'$ à l'aide du Lemme 1, et que $C \mapsto [\![P]\!]^{-1}$ par la règle ($\textsc{Grab}_2$). □

Théorème 6 (Déterminisme)**.** *Pour toute configuration C et tous processus P, P', P'', si $[\![C]\!] \xrightarrow{\tau}{}^{\star} P$, $P \xrightarrow{\tau} P'$ et $P \xrightarrow{\tau} P''$, alors $P' = P''$.*

Démonstration. Par le Lemme 5, on peut se ramener au cas où $P = [\![C]\!]$. En reprenant la preuve de ce même lemme, par analyse de cas sur C, on constate qu'au plus une communication est possible dans le processus traduit $[\![C]\!]$. □

Théorème 7 (Correspondance opérationelle)**.** *Pour toutes configurations C, C', $C \mapsto C'$ si et seulement si $[\![C]\!] \xrightarrow{\tau} [\![C']\!]$.*

Démonstration. Conséquence immédiate des Lemmes 2 et 5. □

Théorème 8. *Pour toute configuration C, nous avons C termine si et seulement si $[\![C]\!] \xrightarrow{\tau}{}^{\star} \xrightarrow{\overline{b}\langle \mathbf{0} \rangle}$.*

Démonstration. La machine abstraite s'arrête exactement sur les configurations de la forme $\lambda' x.M \star []$, dont les encodages sont de la forme $(a(x).[\![M]\!]) \parallel \overline{b}\langle \mathbf{0} \rangle$, qui ont pas de transitions τ, et qui ont une barbe sur b. Inversement, les seules configurations dont la traduction est capable d'émettre sur b sont celles de la forme $\lambda' x.M \star []$. On peut donc conclure par le Théorème 7. □

En tant que fragment du π-calcul d'ordre supérieur [7], HOcore peut naturellement être étendu en ajoutant un opérateur de restriction de nom. Nous ne considérons ici qu'une extension plus réduite n'utilisant que des restrictions globales (i.e., les restrictions ne peuvent être dans des messages, donc elles ne peuvent être répliquées). La syntaxe est étendue de la manière suivante.

$$T ::= \nu a.T \mid P$$

L'extension du LTS est immédiate : les seules transitions autorisées pour les termes $\nu a.T$ sont celles de T dont les noms ne mentionnent pas a. Nous notons $\mathtt{n}(\alpha)$ les noms de canaux de α.

$$\frac{T \xrightarrow{\alpha} T' \qquad a \notin \mathtt{n}(\alpha)}{\nu a.T \xrightarrow{\alpha} \nu a.T'}$$

Intuitivement, $\nu a.P$ correspond au processus P auquel on interdit de communiquer sur a avec l'extérieur (en émission comme en réception) : le nom de canal a est connu de lui seul, toutes les communications sur a ne peuvent donc avoir lieu qu'à l'intérieur de P.

Théorème 9. *Soit $\Omega \triangleq \nu a.\overline{a}\langle a(x).(\overline{a}\langle x \rangle \parallel x) \rangle \parallel a(x).(\overline{a}\langle x \rangle \parallel x)$. Pour tout λ-terme M et toute pile π, on a $\nu a.\nu c.[\![M \star \pi]\!] \simeq \Omega$ si et seulement si $M \star \pi$ ne termine pas.*

Démonstration. Ω est un processus divergent, dont la seule transition est $\Omega \xrightarrow{\tau} \Omega$. Par conséquent, si $M \star \pi$ ne termine pas, alors $\nu a.\nu c.[\![M \star \pi]\!]$ lui est équivalent, puisqu'il ne pourra jamais émettre sur son seul nom libre (b). Inversement, si $M \star \pi$ termine, alors $\nu a.\nu c.[\![M \star \pi]\!]$ finira par émettre sur b, ce qui le distingue du processus Ω. □

Corollaire 10. *L'équivalence contextuelle est indécidable dans HOcore avec deux restrictions globales.*

Un raisonnement similaire permet d'obtenir l'indécidabilité de l'équivalence contextuelle faible dans HOcore avec deux restrictions globales ; on peut même utiliser dans ce cas le processus vide, $\mathbf{0}$, plutôt que le processus divergent Ω (notons cependant que pour cette preuve, on n'a pas la possibilité d'encoder la pile vide par un processus arbitraire ; en particulier, le processus vide ne conviendrait pas : il est nécessaire d'avoir une observable visible lorsque le fond de pile est atteint). En contrepartie, dans le cas fort, le fond de pile peut être traduit par $\mathbf{0}$ en effectuant la comparaison avec Ω.

4.2. Opérateurs de contrôle

Nous ajoutons maintenant les opérateurs de contrôle (call-cc). Rappelons les deux règles de réduction de la KAM définissant ces opérateurs :

$$\mathtt{cc} \star M :: \pi \mapsto M \star k_\pi :: \pi \qquad \text{(CallCC)}$$
$$k_\pi \star M :: \pi' \mapsto M \star \pi \qquad \text{(Restore)}$$

Etant donné un processus P, on définit

$$K(P) \triangleq c(s_0).(s_0 \parallel a(u).c(\_).(u \parallel \overline{c}\langle P\rangle)).$$

Les deux opérateurs sont alors traduits comme suit :

$$\begin{aligned}[\![\mathtt{cc}]\!] &\triangleq c(s_0).(s_0 \parallel c(s).a(u).(u \parallel \overline{c}\langle\overline{a}\langle K(s)\rangle \parallel \overline{c}\langle s\rangle\rangle))\\ [\![k_\pi]\!] &\triangleq K([\![\pi]\!])\end{aligned}$$

Cet encodage nécessite trois transitions pour simuler les règles (CallCC) et (Restore) de la KAM avec opérateurs de contrôle. Comme précédemment pour la règle (Grab), on introduit donc quatre configurations intermédiaires et artificielles dans la KAM, dont la sémantique est définie par les six règles suivantes.

$$\mathtt{cc} \star \pi \mapsto \mathtt{cc}_1 \star \pi \qquad (\text{CallCC}_1)$$
$$\mathtt{cc}_1 \star M :: \pi \mapsto \mathtt{cc}_2 \star M :: \pi \qquad (\text{CallCC}_2)$$
$$\mathtt{cc}_2 \star M :: \pi \mapsto M \star k_\pi :: \pi \qquad (\text{CallCC}_3)$$

$$k_\pi \star \pi' \mapsto k^1_\pi \star \pi' \qquad (\text{Restore}_1)$$
$$k^1_\pi \star M :: \pi' \mapsto k^2_\pi \star M :: \pi' \qquad (\text{Restore}_2)$$
$$k^2_\pi \star M :: \pi' \mapsto M \star \pi \qquad (\text{Restore}_3)$$

Ces quatre configurations s'encodent comme suit dans HOcore.

$$\begin{aligned}[\![\mathtt{cc}_1 \star \pi]\!] &\triangleq [\![\pi]\!] \parallel c(s).a(u).(u \parallel \overline{c}\langle\overline{a}\langle K(s)\rangle \parallel \overline{c}\langle s\rangle\rangle)\\ [\![\mathtt{cc}_2 \star M :: \pi]\!] &\triangleq \overline{a}\langle[\![M]\!]\rangle \parallel a(u).(u \parallel \overline{c}\langle\overline{a}\langle K([\![\pi]\!])\rangle \parallel \overline{c}\langle[\![\pi]\!]\rangle\rangle)\\ &= \overline{a}\langle[\![M]\!]\rangle \parallel a(u).(u \parallel \overline{c}\langle\overline{a}\langle[\![k_\pi]\!]\rangle \parallel \overline{c}\langle[\![\pi]\!]\rangle\rangle)\\ &= \overline{a}\langle[\![M]\!]\rangle \parallel a(u).(u \parallel \overline{c}\langle[\![k_\pi :: \pi]\!]\rangle)\\ [\![k^1_\pi \star \pi']\!] &\triangleq [\![\pi']\!] \parallel a(u).c(\_).(u \parallel \overline{c}\langle[\![\pi]\!]\rangle)\\ [\![k^2_\pi \star M :: \pi']\!] &\triangleq \overline{c}\langle[\![\pi']\!]\rangle \parallel c(\_).([\![M]\!] \parallel \overline{c}\langle[\![\pi]\!]\rangle)\\ &= \overline{c}\langle[\![\pi']\!]\rangle \parallel c(\_).\,[\![M \star \pi]\!]\end{aligned}$$

On vérifie alors que les processus traduits correspondants ont exactement les transitions suivantes.

$$\begin{aligned}&[\![\mathtt{cc} \star M :: \pi]\!] \xrightarrow{\tau} [\![\mathtt{cc}_1 \star M :: \pi]\!] \xrightarrow{\tau} [\![\mathtt{cc}_2 \star M :: \pi]\!] \xrightarrow{\tau} [\![M \star k_\pi :: \pi]\!]\\ &[\![k_\pi \star M :: \pi']\!] \xrightarrow{\tau} [\![k^1_\pi \star M :: \pi']\!] \xrightarrow{\tau} [\![k^2_\pi \star M :: \pi']\!] \xrightarrow{\tau} [\![M \star \pi]\!]\end{aligned}$$

Les preuves des lemmes 1, 2, 4 et 5 ainsi que des théorèmes 6 et 7 s'étendent sans difficulté.

Le lecteur attentif aura remarqué que la traduction de l'opérateur cc effectue d'abord une réception $c(s)$ avant la réception $a(u)$. Ce choix est purement esthétique : changer l'ordre des réceptions est tout à fait possible, mais ne permet pas d'utiliser $[\![k_\pi :: \pi]\!]$ dans la traduction de $\mathtt{cc}_2$, la rendant moins succincte.

4.3. Version synchrone

Nous montrons maintenant qu'il est possible d'obtenir une traduction de la KAM en utilisant un seul nom, si le calcul considéré est *synchrone*. Une version synchrone de HOcore ne modifie que

l'émission de message, remplaçant la construction $\overline{a}\langle P\rangle$ par $\overline{a}\langle P\rangle.Q$. Cette dernière émet toujours un message sur a transportant P. En revanche, dès que le message est reçu, le processus Q, appelé *continuation*, est lancé : la règle de réduction intuitive devient

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

Formellement, cela se traduit dans le LTS par une simple modification de la règle axiome Out :

$$\frac{}{\overline{a}\langle P\rangle.Q \xrightarrow{\overline{a}\langle P\rangle} Q}\ \textsc{Out}$$

La traduction de la KAM (étendue) est donnée ci-dessous. Nous traduisons désormais les piles comme étant des messages synchrones : le contenu du message étant la tête de la pile et la continuation du message la queue de la pile. Nous traduisons ainsi une pile $1 :: 2 :: 3 :: []$ par $\overline{a}\langle 1\rangle.\overline{a}\langle\overline{a}\langle 2\rangle.\overline{a}\langle\overline{a}\langle 3\rangle.\overline{a}\langle\overline{b}\langle\mathbf{0}\rangle\rangle\rangle\rangle$.

Au sein d'une configuration, comme dans le cas asynchrone, la pile sera de plus encapsulée à l'intérieur d'un message additionel sur a.

Comme ci-dessus, pour tout processus P, nous définissons

$$K(P) \triangleq a(s_0).(s_0 \parallel a(u).a(\_).(u \parallel \overline{a}\langle P\rangle)).$$

(Par convention, nous notons $\overline{a}\langle P\rangle$ les messages $\overline{a}\langle P\rangle.\mathbf{0}$ dont la continuation est vide). Nous définissons alors la traduction synchrone comme suit.

$$\begin{aligned}
[\![[]]\!] &\triangleq \overline{b}\langle\mathbf{0}\rangle\\
[\![M :: \pi]\!] &\triangleq \overline{a}\langle[\![M]\!]\rangle.\overline{a}\langle[\![\pi]\!]\rangle\\
[\![MN]\!] &\triangleq a(s).([\![M]\!] \parallel \overline{a}\langle\overline{a}\langle[\![N]\!]\rangle.\overline{a}\langle s\rangle\rangle)\\
[\![\lambda x.M]\!] &\triangleq a(s).(a(x).\,[\![M]\!] \parallel s)\\
[\![x]\!] &\triangleq x\\
[\![M \star \pi]\!] &\triangleq [\![M]\!] \parallel \overline{a}\langle[\![\pi]\!]\rangle\\
[\![\mathtt{cc}]\!] &\triangleq a(s_0).(s_0 \parallel a(u).a(s).(u \parallel \overline{a}\langle\overline{a}\langle K(s)\rangle.\overline{a}\langle s\rangle\rangle))\\
[\![k_\pi]\!] &\triangleq K([\![\pi]\!])
\end{aligned}$$

A nouveau, les preuves de la section 4.1 s'adaptent sans difficulté. L'équivalence contextuelle est donc indécidable dans HOcore synchrone avec une seule restriction de nom globale.

5. Conclusion

Nous avons présenté deux traductions directes de la machine abstraite de Krivine avec opérateurs de contrôle dans HOcore, utilisant un nom libre dans le cas synchrone, et deux dans le cas asynchrone. Cela nous a permis d'affiner la borne sur le nombre de restrictions globales de noms de canaux suffisant pour obtenir l'indécidabilité des équivalences contextuelles fortes et faible (en l'absence de restriction de nom, l'équivalence contextuelle forte est décidable que le calcul soit synchrone ou asynchrone [5]—le cas faible est ouvert).

Nous pouvons remarquer que les trois fonctionnalités fondamentales utilisées pour traduire la KAM sont les suivantes. Les deux premières portent sur l'ordre supérieur : les messages transportent des processus, et la réception effectue une substitution identique à celle du λ-calcul. La troisième fonctionnalité sur laquelle nous nous appuyons porte sur le contrôle de l'évaluation : nous devons

distinguer entre la tête de la pile et le reste de la pile. Pour ce faire, nous pouvons utiliser deux noms différents (version asynchrone), ou séquentialiser les émissions (version synchrone) et n'utiliser ainsi qu'un seul nom. Nous ne pensons pas qu'il soit possible d'obtenir une traduction de la KAM avec un seul nom dans un calcul asynchrone.

La relative simplicité de notre traduction, et le fait qu'elle permette de traiter immédiatement les opérateurs de contrôle, nous laisse espérer qu'elle permettra une meilleure compréhension de l'expressivité du calcul HOcore, et peut-être, à terme, de résoudre la question de la décidabilité de l'équivalence contextuelle faible—dans le calcul sans restriction de nom.

Références

[1] J. Aranda, F. D. Valencia, and C. Versari. On the expressive power of restriction and priorities in ccs with replication. In L. Alfaro, editor, *Foundations of Software Science and Computational Structures*, volume 5504 of *Lecture Notes in Computer Science*, pages 242–256. Springer Berlin Heidelberg, 2009.

[2] S. Boulier and A. Schmitt. Formalisation de hocore en coq. In *Actes des 23èmes Journées Francophones des Langages Applicatifs*, Jan. 2012.

[3] N. Busi, M. Gabbrielli, and G. Zavattaro. On the expressive power of recursion, replication and iteration in process calculi. *Mathematical Structures in Computer Science*, 19(6) :1191–1222, Dec. 2009.

[4] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3) :199–207, 2007.

[5] I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness and decidability of higher-order process calculi. *Information and Computation*, 209(2) :198–226, Feb. 2011. Extended abstract presented at *Logic in Computer Science (LICS)*, 2008.

[6] R. Milner and D. Sangiorgi. Barbed bisimulation. In *19th ICALP*, volume 623 of *LNCS*, pages 685–695. Springer Verlag, 1992.

[7] D. Sangiorgi and D. Walker. *The π-calculus : a Theory of Mobile Processes*. Cambridge University Press, 2001.

Analyse de dépendances et correction de réseaux de preuve

Marc Bagnol[1] & Amina Doumane[2] & Alexis Saurin[3]

1 : IML, Université d'Aix-Marseille, bagnol@iml.univ-mrs.fr
2 : PPS, Université Paris Diderot & École centrale Paris, amina.doumane@student.ecp.fr
3 : PPS, CNRS, Université Paris Diderot & INRIA, alexis.saurin@pps.univ-paris-diderot.fr

Ce travail a bénéficié du soutien du projet ANR portant la référence ANR-10-BLAN-0213.

1. Introduction

Les fruits de Curry-Howard. Depuis la mise en évidence via la correspondance de Curry-Howard [21], des relations entre démonstrations mathématiques et programmes informatiques, théorie de la programmation et théorie de la démonstration s'enrichissent mutuellement. Parmi les nombreux allers-retours fructueux entre preuves et programmes, la logique linéaire [13] tient une place exemplaire. Girard a introduit le système F [11] alors qu'il travaillait sur l'arithmétique du second ordre. Ce système, redécouvert de manière indépendante par Reynolds [27], fournit un λ-calcul polymorphe (ou λ-calcul du second-ordre).

Une quinzaine d'années plus tard, Girard mit en évidence la sémantique cohérente du système F [12] qui élabore sur la sémantique stable de Berry [2]. Les espaces cohérents ont conduit à la décomposition de l'implication intuitionniste à l'origine de la logique linéaire [13] : $A \Rightarrow B = !\,A \multimap B$. Cette décomposition linéaire, observée dans la sémantique du système F, se reflète syntaxiquement en un système de déduction bien structuré. S'est alors ouvert un champ original où se sont renouvelés les points de vue sur les preuves et les programmes.

Du côté logique, sont apparus les réseaux de preuve [13, 15, 7], les sémantiques interactives (géométrie de l'interaction, sémantique de jeux, ludique), les logiques polarisées [24], allégées [16] et plus récemment différentielles [10].

Du côté programmation, l'étude de la linéarité a eu des conséquences sur de nombreux problèmes : optimalité et partage [18, 22], interprétations calculatoires de la logique classique [14, 6, 24], systèmes de type fournissant des bornes de complexité [23], lien entre focalisation et langages logiques [1, 25], interprétation logique du filtrage [28, 4].

De la logique linéaire aux réseaux de preuve. La décomposition linéaire mentionnée plus haut fait donc apparaître des connecteurs, dits exponentiels (!, ?), qui contrôlent l'usage des hypothèses (via les règles structurelles de contraction et affaiblissement). Une conséquence importante du contrôle des règles structurelles réside dans le fait que les diverses présentations des inférences pour la conjonction et la disjonction, équivalentes en logique classique, ne le sont plus dans le cadre linéaire. On voit alors apparaître deux disjonctions et deux conjonctions, deux constantes logiques pour le vrai et deux pour le faux, répartis en un groupe dit multiplicatif ($\parr$, $\otimes$, 1, $\bot$) et un groupe additif ($\oplus$, &, $\top$, 0). La linéarité permet également de récupérer une négation involutive ($A^{\bot\bot} = A$) et les lois de de Morgan permettent alors de restreindre la négation aux formules atomiques tout en considérant des séquents dont toutes les formules sont à droite. LL possède une bonne notion de fragments logiques : on

peut ainsi considérer le fragment restreint aux connecteurs multiplicatifs (avec ou sans les constantes) appelé MLL ou bien le fragment multiplicatif exponentiel, appelé MELL.

Les réseaux de preuve constituent l'une des innovations les plus originales de la logique linéaire. Fondés de manière essentielle sur la linéarité, les réseaux de preuve constituent une syntaxe graphique pour les preuves constituant des objets de preuve très canoniques [1] et l'élimination des coupures y est particulièrement élégante.

L'élégance et la simplicité des réseaux sont particulièrement frappants dans le fragment multiplicatif et sans unités de la logique linéaire (MLL) sur lequel l'article va se concentrer. Il s'agit d'ailleurs plus d'une simplification de présentation qu'une véritable restriction puisque nos résultats s'étendent tous à MELL de manière directe, capturant ainsi le lambda-calcul typé.

Réseaux de preuve et correction logique. Abandonnant une notation séquentielle au profit d'une représentation de la preuve comme un graphe, les réseaux de preuve s'éloignent de l'idée habituelle qu'on se fait d'une démonstration (typiquement un raisonnement qui se déroule progressivement, de manière ordonnée, de son point de départ jusqu'à sa conclusion, structuré par des lemmes intermédiaires).

L'une des conséquences de cette non-séquentialité est que s'ils contiennent des erreurs de raisonnement (ou paralogismes), celles-ci ne sont pas forcément faciles à détecter, contrairement à la plupart des systèmes de déduction (systèmes de Hilbert, déduction naturelle, calcul des séquents, ...) où la correction logique d'une preuve se vérfie de manière purement locale. On parlera de structure de preuve pour un objet qui ressemble à un réseau mais n'est pas forcément logiquement correct.

C'est le prix à payer pour avoir la syntaxe graphique et les bonnes propriétés des réseaux : on passe en effet d'objets inductifs (les arbres des preuves en calcul des séquents par exemple) à des objets de nature plus géométrique (les structures de preuve) dont les propriétés comme l'acyclicité ou la connexité ne sont plus locales mais sont globales.

Pourtant, les démonstrations sont avant tout (et même avant d'être des objets calculatoires !) des objets qui servent à se forger une conviction et à la transmettre. Reformulé en ces termes, le problème de la correction des structures de preuve se résume à ceci : si l'on communique à quelqu'un une structure de preuve R, il doit disposer des moyens pour être sûr que la structure est logiquement correcte et qu'elle ne contient pas un raisonnement erroné [2]. Il faudra éventuellement fournir à cette personne, en même temps qu'on lui communique la structure de preuve, un certificat de correction de cette structure de preuve [3].

On souhaite donc disposer de conditions sur les réseaux, a priori de nature graphique et géométrique, assurant de la correction des réseaux, c'est-à-dire qui permette de discriminer les réseaux de preuve qui sont logiquement corrects de ceux qui contiennent des erreurs de raisonnement.

C'est toute la problématique de la correction des réseaux de preuve auquel cet article se veut une contribution : nous proposons un nouveau critère particulièrement simple qui permet de déterminer si une structure de preuve représente bien une preuve. Notre critère simplifie un critère récemment publié par Mogbil et Naurois.

Organisation de l'article. On commence par rappeler les bases de la logique linéaire et des réseaux de preuve. On rappelle le critère de correction de Danos et Régnier. On considère ensuite le critère

1. Contrairement aux preuves du calcul des séquents qui contiennent beaucoup d'information non pertinente sur l'ordonnancement des règles d'inférence.

2. Ou encore, vu sous l'angle calculatoire, qu'il s'agit d'un programme bien typé...

3. L'idée la plus simple est bien sûr de transmettre une preuve séquentialisée de la structure R, mais on voit vite les limites de cette idée : non seulement parce qu'elle nous fait nous reposer sur le calcul des séquents mais aussi parce que le réseau qu'on souhaite transmettre peut résulter d'une élimination des coupures et dans ce cas on ne dispose pas forcément d'une version séquentialisée de la preuve.

de contractilité qui nous servira dans la dernière partie de l'article. On en vient ensuite au cœur de l'article qui est la notion de graphe de dépendance. On commence par présenter le critère de Mogbil et Naurois puis on discute du rôle de l'interrupteur considéré dans le critère. On élimine la dépendance de notre critère aux interrupteurs en deux étapes. On montre tout d'abord qu'on peut définir une structure de graphe de dépendance sur la structure de preuve elle-même et non plus sur ses graphes de corrections ce qui nous donne une première variante de Mogbil et Naurois. On achève de se libérer des interrupteurs en proposant le critère DepGraph. Finalement, on analyse les relations entre les graphes de dépendance introduits dans cet article et les graphes de dépendance de Mogbil et Naurois.

2. Logique linéaire et réseaux de preuve

MLL. Nous nous restreindrons dans la suite de l'article au fragment multiplicatif de la logique linéaire sans constantes, noté MLL. Les formules de MLL sont construites à partir de la grammaire suivante :

$$\boxed{A, B := X \mid X^{\perp} \mid A \otimes B \mid A \parr B \quad (X \in \mathcal{V})}$$

MLL se présente d'habitude en calcul des séquents : un séquent de MLL est une liste finie non orientée de formules de MLL, noté $\vdash \Gamma$ et une preuve est un arbre dont les nœuds sont étiquetés par $(ax), (cut), (\otimes), (\parr)$ et dont les arêtes sont étiquetées par des séquents, suivant les règles ci-dessous :

$$\frac{}{\vdash A, A^{\perp}}\ (ax) \qquad \frac{\vdash \Gamma, A \qquad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}\ (cut) \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\ (\otimes) \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ (\parr)$$

Remarque. *Si on peut voir MLL comme un fragment de LL, on peut également le voir comme un restriction du calcul des séquents de LK où on interdit les règles structurelles de contraction et d'élimination. Le $\otimes$ doit alors être lu comme la conjonction habituelle $\wedge$ et le $\parr$ comme la disjonction $\vee$. Ce parallèle pourra être utile au lecteur qui n'est pas familier avec la logique linéaire.*

Structures et réseaux de preuve. Les structures de preuve sont une nouvelle syntaxe pour la logique linéaire. Comme nous ne nous restreignons qu'au fragment MLL, nous allons présenter uniquement les structures de preuve pour ce fragment. Il existe dans la littérature des notions de structures de preuve qui correspondent aux autres fragments de la logique linéaire.

Définition 2.1. ***Structure de preuve.*** *On appelle structure de preuve un graphe orienté fini dont les sommets (appelés aussi nœuds) sont étiquetés soit par des connecteurs de la logique linéaire $(\parr, \otimes)$ soit par* ***ax*** *(pour axiome) ou* ***cut*** *(pour coupure) ou* ***c*** *(pour conclusion) et les arêtes sont étiquetées par des formules de la logique linéaire. Les sommets et les arêtes vérifient de plus les propositions suivantes :*

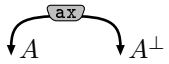
- *Les nœuds étiquetés par $\otimes$ (resp. $\parr$) ont deux prémisses et une conclusion. Si l'étiquette de la première prémisse est A et celle de la deuxième prémisse est B alors l'étiquette de la conclusion est $A \otimes B$ (resp. $A \parr B$) ;*
- *Les nœuds étiquetés par* ***ax*** *n'ont aucune prémisse et ont deux conclusions. Si l'étiquette de la première conclusion est A alors celle de la deuxième conclusion est $A^{\perp}$;*
- *Les nœuds étiquetés par* ***cut*** *ont deux prémisses et n'ont aucune conclusion. Si l'étiquette de la première prémisse est A alors celle de la deuxième prémisse est $A^{\perp}$;*
- *Les nœuds étiquetés par* ***c*** *ont une prémisse et n'ont aucune conclusion*[4].

4. Pour alléger la représentation graphique des réseaux, on laissera ces liens implicites dans les figures, qui seront représentés comme des arêtes pendantes.

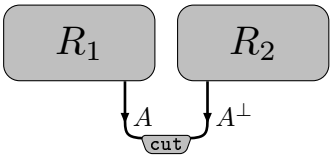
À chaque preuve dans MLL on va associer une structure de preuve. On appellera *réseaux de preuve* le sous-ensemble des structures de preuve qui proviennent de dérivations MLL.

Définition 2.2. ***Réseaux de preuve.*** *On définit par induction sur la dernière règle utilisée la structure de preuve correspondant à une preuve de MLL.*

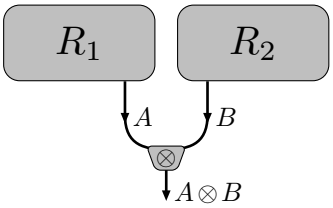
- *règle* ***ax*** *: le réseau de preuve correspondant à $\vdash A, A^\perp$ est le graphe contenant un nœud ax dont les arêtes sortantes-étiquetées par A et $A^\perp$ sont reliées à ces nœuds conclusion :*



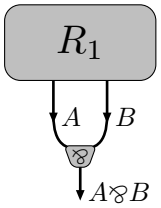
- *règle* ***cut*** *: si R_1 et R_2 sont les réseaux de preuve associés aux deux prémisses de la règle, le réseau de preuve associé à la preuve complète est obtenu en ajoutant un nœud* ***cut*** *entre les arêtes correspondant aux occurrences de formules impliquées dans la règle :*



- *règle $\otimes$: si R_1 et R_2 sont les réseaux de preuve associés aux deux prémisses de la règle, le réseau de preuve associé à la preuve complète est obtenu en ajoutant un nœud $\otimes$ entre les arêtes correspondant aux occurrences de formules impliquées dans la règle, et en reliant l'arête sortante à un nœud conclusion :*



- *règle $⅋$: si R_1 est le réseau de preuve associé à la prémisse de la règle, le réseau de preuve associé à la preuve complète est obtenu en ajoutant un nœud $⅋$ entre les arêtes correspondant aux occurrences de formules impliquées dans la règle, et en reliant l'arête sortante à un nœud conclusion :*



Le graphe de la figure 1 est bien une structure de preuve, pourtant il ne peut être associé à une preuve dans MLL. Une structure de preuve ne correspond donc pas nécessairement à une preuve en calcul des séquents et une telle structure est dite non séquentialisable. Pour distinguer les structures de preuve séquentialisables — les réseaux de preuve — de celles qui ne le sont pas, il existe de nombreux résultats décrivant des méthodes pour faire cette distinction, sous le nom de *critères de correction*.

Remarque. *Dans la suite de l'article, nous ne considérerons plus que des réseaux sans coupures. La coupure se comporte en effet exactement comme le $\otimes$ du point de vue de la correction et n'introduit donc ni difficulté ni intérêt particuliers.*

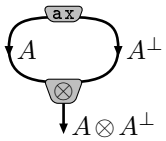


FIGURE 1 – UNE STRUCTURE DE PREUVE QUI N'EST PAS UN RÉSEAU.

3. Critères de correction

Plusieurs critères de correction pour les structures de preuve ont été introduits pour distinguer, parmi les structures de preuve, celles qui sont des réseaux de preuve de celles qui ne sont pas correctes. On compte notamment le critère original des longs voyages (LV) [13], le critère de Danos-Regnier (DR) [7], le critère des contre-preuves (CP) [3, 17], le critère de contractilité [5], le critère de parsing [19, 20] et, plus récemment apparu, le critère de Mogbil-Naurois [8].

Dans cette section, nous allons présenter les critères Danos-Regnier et contractilité qui seront utilisés ultérieurement. Le critère de Mogbil et Naurois sera introduit dans la section suivante. Le reste de l'article consistera en l'introduction d'un nouveau critère de correction, DepGraph, basée sur le critère de Mogbil et Naurois.

3.1. Critère de Danos-Regnier

Définition 3.1. ***Interrupteur.*** *On appelle interrupteur (ou switching) d'une structure de preuve le choix pour chacun de ses nœuds $\parr$ de l'une des deux prémisses. De chaque nœud $\parr$, on dit qu'il est switché à droite (resp. à gauche), si on a choisi la prémisse droite (resp. gauche).*

Définition 3.2. ***Graphe de correction.*** *On appelle graphe de correction $\mathcal{S}(R)$ d'une structure de preuve R et d'un interrupteur $\mathcal{S}$ le graphe non-orienté dont :*

1. *Les sommets sont ceux de la structure de preuve.*
2. *Les arêtes sont les mêmes que celles de la structure de preuve où on a supprimé l'arête gauche (resp. droite) d'un nœud $\parr$ s'il a été switché à droite (resp. à gauche).*
3. *Les étiquettes sont les mêmes que dans la structure de preuve.*

Définition 3.3. ***Critère de Danos-Regnier(DR).*** *On dit qu'une structure de preuve vérifie le critère de Danos-Regnier si tous les graphes de correction associés à des interrupteurs sont connexes et acycliques. On dit aussi qu'elle est DR-correcte.*

Théorème 3.4. *Un structure de preuve est un réseau de preuve si et seulement si elle est DR-correcte.*

Le critère de Danos-Regnier est un critère qui ne fait pas référence directement à la séquentialisabilité d'une structure de preuve en une preuve du calcul des séquents. La simplicité de son énoncé et son élégance en ont fait l'un des critères les plus fameux. En revanche, il n'est pas efficace pour tester la correction d'une structure de preuve puisqu'il faut vérifier la connexité et l'acyclicité des 2^n graphes de correction, où n est le nombre de $\parr$ de la structure, ce qui est exponentiellement chronophage. On ne peut pas espérer améliorer aisément cette borne en se restreignant à un sous-ensemble d'interrupteurs bien choisis : on peut en effet montrer qu'il existe des structures de preuves ayant un nombre arbitrairement grand de $\parr$, dont tous les graphes de correction sont connexes et acycliques sauf un. À moins de disposer d'une information très spécifique sur la topologie du réseau (par exemple si le graphe est planaire, on peut se restreindre à ne tester que deux interrupteurs pour décider de la correction d'une structure de preuve [26]).

Obtenir des algorithmes efficaces pour tester la correction d'une structure de preuve est donc un enjeu important pour rendre utilisable cette représentation des preuves et, logiquement, de nombreuses recherches ont été consacrées à produire des critères dont les complexités étaient de plus en plus faibles. Ainsi, par exemple, le critère de contractilité [5] a-t-il une complexité quadratique tandis que le critère de parsing [20] peut être testé en temps linéaire. Dans la section suivante, nous présentons le critère de contractilité qui nous servira dans la suite de notre développement.

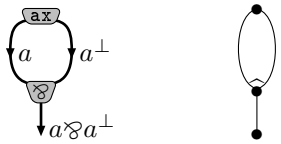
3.2. Critère de contractilité

Le critère de Danos-Regnier reposait déjà sur des propriétés topologiques des structures de preuve, plus exactement de leur graphes de correction. Le critère que l'on introduit maintenant exprime quant à lui directement une propriété topologique de la structure de preuve (ou plus exactement du graphe apparié qui lui est associé, qui contient juste l'information nécessaire pour distinguer les arêtes prémisses d'un $\parr$ des autres arêtes de la structure) en terme de contractilité.

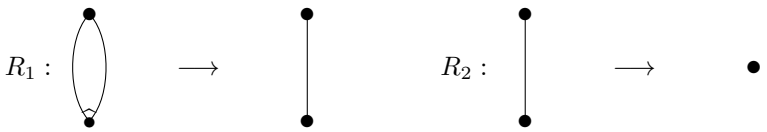
Définition 3.5. ***Graphe apparié.*** *Un graphe apparié est la donnée d'un graphe $G = (V, E)$ et d'un ensemble $P(G)$ de paires (non-orientées) d'éléments de E qui ont au moins une extrémité commune. L'extrémité commune de deux arêtes appariées est appelée un nœud apparié.*

Définition 3.6. $C(R)$**.** *Soit R une structure de preuve. Le graphe apparié qui lui est associé, qu'on note $C(R)$ est le graphe R muni de l'ensemble des paires d'arêtes qui sont prémisses d'un nœud $\parr$.*

Exemple. *On donne ci-dessous l'unique réseau de preuve $R_{a \parr a^\perp}$ pour le séquent $\vdash a \parr a^\perp$ et le graphe apparié $C(R_{a \parr a^\perp})$ qui lui est associé (les arêtes appariées sont distinguées par un $\frown$) :*



Définition 3.7. ***Règles de contraction.*** *On définit les règles de réécriture sur les graphes appariés suivantes (on notera que dans la règle R_2, on demande que les deux nœuds soient distincts) :*



Définition 3.8. ***Contractilité.*** *On dit qu'une structure de preuve R est contractile si $C(R) \to^* \bullet$.*

La contractilité caractérise les réseaux de preuve et fournit donc un critère de correction :

Théorème 3.9. *Une structure de preuve est un réseau de preuve si et seulement si elle est contractile.*

4. Le critère du graphe de dépendance

On introduit maintenant le critère de Mogbil et Naurois qui repose :

1. sur l'existence d'un interrupteur dont le graphe de correction est connexe et acyclique et
2. sur le fait que le *graphe de dépendance* construit à partir de cet interrupteur soit un graphe acyclique et possédant une source.

On cherche ensuite à s'abstraire complètement de l'interrupteur, ce qui se fait en deux étapes. Dans la section 4.2, on commence par définir une nouvelle notion de graphe de dépendance qui ne dépend pas du choix d'un interrupteur, on obtient ainsi une condition nécessaire pour qu'une structure de preuve soit séquentialisable. Finalement, on se passera complètement de l'interrupteur dans la section 4.3 où l'on analysera l'utilité véritable de la première partie du critère dont nous montrerons qu'elle peut être remplacée par une condition nécessaire beaucoup plus simple qui ne dépend pas des interrupteurs. La conjonction de ces deux conditions nécessaires fournit en fait une condition suffisante comme nous le montrons dans la section 4.4. On conclut la section en comparant en 4.5 les graphes de dépendance à la Mogbil-Naurois et les graphes de dépendance que nous venons d'introduire.

4.1. Le critère de Mogbil et Naurois

Le critère de Mogbil et Naurois a été introduit pour montrer que la correction des structures de preuve MLL était NL-complète[5].

Définition 4.1. ***Chemin élémentaire.*** *Un chemin dans un graphe non-orienté sera dit élémentaire s'il ne passe pas deux fois par la même arête.*

Définition 4.2. ***Graphe de dépendance d'un graphe de correction.*** *Le graphe de dépendance d'un graphe de correction provenant d'un interrupteur $\mathcal{S}$ d'une structure de preuve R, noté $D(\mathcal{S}, R)$ est un graphe orienté (S, A) défini comme suit :*

- *L'ensemble des nœuds S est constitué de l'ensemble des conclusions des nœuds $\parr$ de R et d'un nœud s.*
- *Soit x un nœud $\parr$ dans R, x_d et x_g ses prémisses droite et gauche respectivement dans R.*
  - *Il y a une arête $(s \to x)$ dans A s'il existe un chemin élémentaire $x_g, \ldots, x_d$ dans $\mathcal{S}(R)$ qui ne passe par aucun nœud $\parr$.*
  - *Soit y un autre nœud $\parr$ de R. Il y a une arête $(y \to x)$ s'il existe un chemin élémentaire $x_g, \ldots, x_d$ dans $\mathcal{S}(R)$ qui contient y.*

Définition 4.3. ***Graphe** SDAG.* *Un graphe G est dit SDAG si :*

- *il est acyclique ;*
- *il contient un nœud s, nommé nœud source, tel que, pour chaque nœud n de G différent de s, il existe un chemin de s vers n.*

Définition 4.4. ***Critère de Mogbil-Naurois.*** *Une structure de preuve vérifie le critère de Mogbil-Naurois (MN) si, et seulement si, il existe un interrupteur $\mathcal{S}$ tel que :*

- *$D(\mathcal{S}, R)$ est SDAG de source s ;*
- *$\mathcal{S}$ est connexe et acyclique.*

Theorem 4.5. *Une structure de preuve est séquentialisable si, et seulement si, elle vérifie (MN).*

Les figures 2 et 3 illustrent comment ce critère discrimine les structures correctes et incorrectes. On remarque sur la figure 3 que le graphe de dépendance dépend de l'interrupteur, on reviendra sur ce point dans la section 4.5.

5. NL désigne la classe des problèmes qui peuvent être décidés en espace logarithmique par une machine de turing non déterministe.

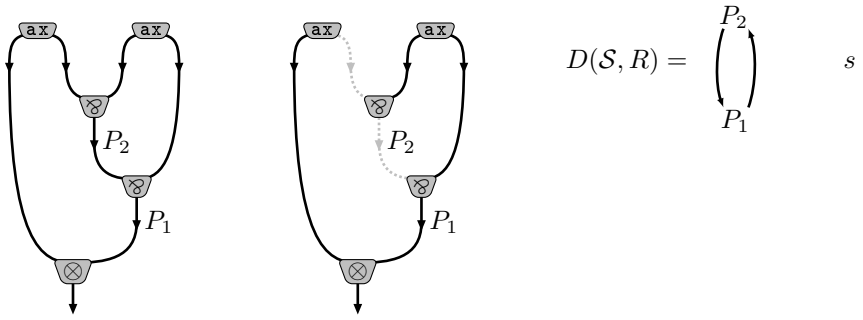


FIGURE 2 – UNE STRUCTURE INCORRECTE R, UN INTERRUPTEUR $\mathcal{S}$ DE R ET LE GRAPHE DE DÉPENDANCE (CYCLIQUE ET SANS SOURCE) ASSOCIÉ.

Par rapport à Danos-Regnier, le critère de Mogbil et Naurois possède une caractéristique troublante : il ne repose que sur l'analyse d'un interrupteur et du graphe de correction induit par cet interrupteur, d'autant que le choix de l'interrupteur est lui-même arbitraire.

Il semble donc naturel de se demander la véritable raison d'être de cet interrupteur : à quoi sert-il ? En a-t-on vraiment besoin ? Nous répondons à ces deux questions dans la suite en énonçant un critère de correction basé sur le graphe de dépendance qui ne dépende pas d'un interrupteur.

4.2. Un graphe de dépendance qui ne dépend pas des interrupteurs

On définit une notion de graphe de dépendance directement à partir d'une structure de preuve et non plus à partir de ses graphes de correction.

Définition 4.6. ***s-chemin dans une structure de preuve.*** *Dans une structure de preuve, un chemin est appelé s-chemin s'il ne passe pas successivement par les deux prémisses d'un même nœud $\parr$ et s'il est élémentaire.*

Définition 4.7. ***Graphe de dépendance d'une structure de preuve.*** *Le graphe de dépendance d'une structure de preuve R, noté $D(R)$ est un graphe orienté (S, A) défini comme suit :*

- *L'ensemble des nœuds S est constitué de l'ensemble des conclusions des nœuds $\parr$ de R et d'un nœud s.*
- *Soit x un nœud $\parr$ dans R, x_d et x_g ses prémisses droite et gauche respectivement dans R.*
  - *Il y a une arête $(s \to x)$ dans A s'il existe un s-chemin $x_g, \ldots, x_d$ dans R qui ne passe par aucun nœud $\parr$.*
  - *Soit y un autre nœud $\parr$ de R. Il y a une arête $(y \to x)$ s'il existe un s-chemin $x_g, \ldots, x_d$ dans R qui contient y.*

Définition 4.8. ***Critère de Mogbil-Naurois modifié.*** *Une structure de preuve vérifie le critère de Mogbil-Naurois modifié (MN') si, et seulement si :*

- *son graphe de dépendance $D(R)$ est SDAG de source s ;*
- *un de ses graphes de correction est connexe et acyclique.*

Les condition (MN') fournissent sont nécéssaires pour qu'une structure soit séquentialisable :

Theorem 4.9. *Une structure de preuve séquentialisable satisfait (MN').*

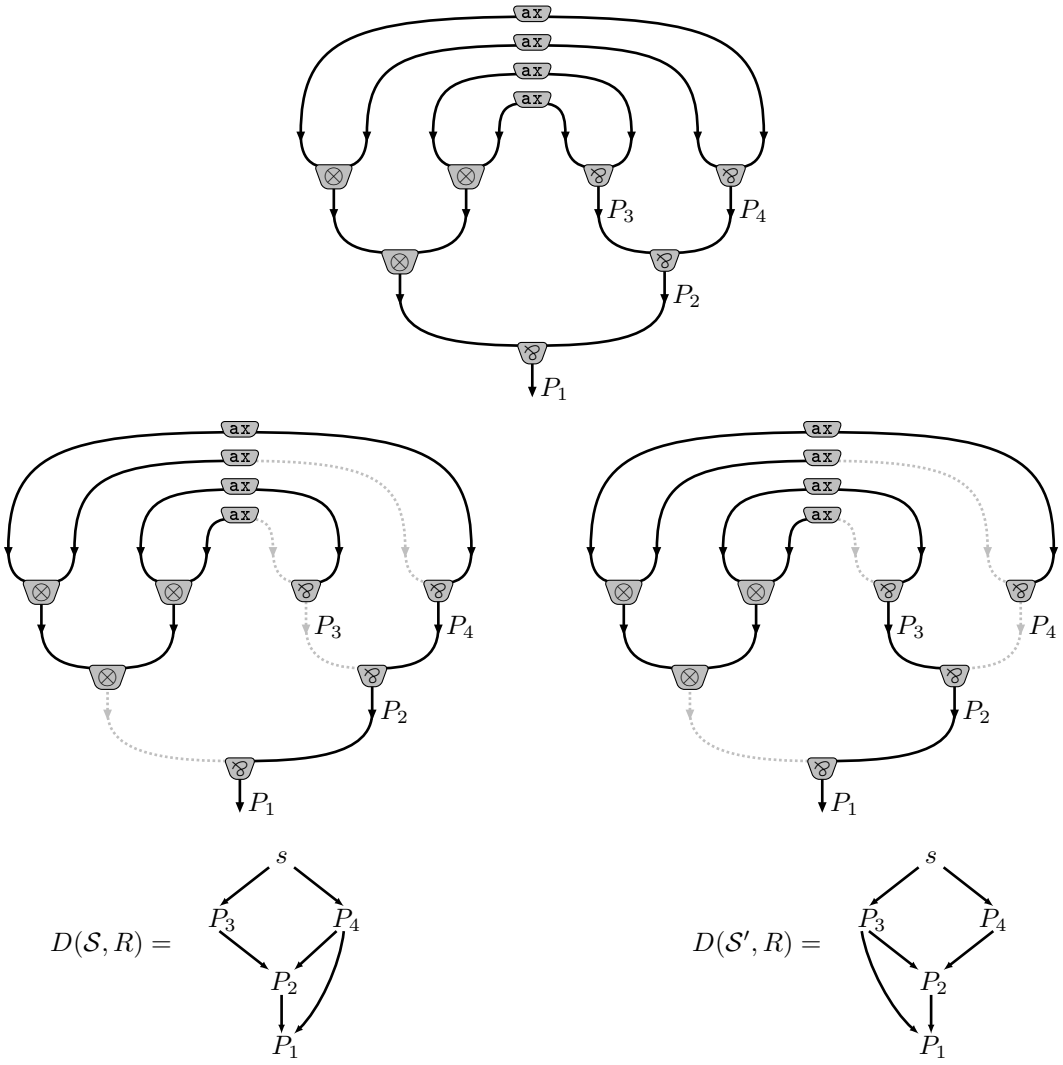


FIGURE 3 – UN RÉSEAU R, DEUX INTERRUPTEURS $\mathcal{S}$ ET $\mathcal{S}'$ DE R ET LES GRAPHES DE DÉPENDANCE ASSOCIÉS.

Pour montrer ce résultat, on s'appuie sur une relation entre les $\parr$ induite par l'ordre dans lequel les inférences sont utilisées dans une des séquentialisations de R qui nous donnera l'acyclicité du graphe de dépendance. Une fois l'acyclicité obtenue, l'existence d'une source découle du fait qu'au plus un nœud ne possède pas de parent.

Définition 4.10. ***Ordre d'introduction.*** *Soit π une preuve MLL. Pour toute règle $\parr$ $\mathbf{r}_F$ introduisant la formule F, on note π_F le sous-arbre de π dont la racine est la prémisse de $\mathbf{r}_F$.*

On définit un ordre sur les formules introduites par une règle $\parr$ dans π, qu'on notera $<_\pi$ (qui correspond informellement à "être introduit au dessus" dans l'arbre de preuve), comme suit :

$$F <_\pi G \text{ si } \mathbf{r}_F \in \pi_G$$

On notera le graphe de cette relation $O^-(\pi)$ et on définit $O(\pi)$ comme le graphe $O^-(\pi)$ auquel on a ajouté un sommet s et une arête $s \to e$ pour tout sommet e de $O^-(\pi)$.

Théorème 4.11. *Soit π une preuve MLL et R le réseau de preuve correspondant. On a :*

$$D(R) \subseteq O(\pi)$$

Démonstration. Voir en annexe. □

Lemme 4.12. *Soit π une preuve MLL. $O(\pi)$ est acyclique.*

Démonstration. Immédiat. □

Lemme 4.13. *Soit G un graphe dirigé acyclique. Si chaque nœud de G a un parent, sauf un seul nœud, noté s, alors G est $SDAG$ de source s.*

On peut maintenant passer à la preuve du théorème 4.9 :

Démonstration. Soit R un réseau de preuve.

- Par Danos-Régnier, il est évident que tout graphe de correction est connexe et acyclique.
- Comme R est un réseau de preuve, il existe une preuve π dont R est le réseau. Par le théorème 4.11, on a que $D(R) \subseteq O(\pi)$. Mais $O(\pi)$ est acyclique par le lemme 4.12. On conclut alors que $D(R)$ est acyclique. D'autre part, comme R est un réseau de preuve, par le critère de Danos-Regnier, tous ses graphes de correction sont connexes. Soit G un graphe de correction de R. Comme il est connexe, les deux prémisses de chaque nœud $\parr$ sont reliées par un chemin dans G, qui est un s-chemin dans R. Il s'en suit que dans $D(R)$, chaque nœud - à part le nœud s- a un parent. Le lemme 4.13 nous assure donc que $D(R)$ est $SDAG$.

□

Remarque. *Le critère MN' n'est pas seulement une condition nécessaire : il s'agit bien d'un critère de correction [9]. Nous n'allons cependant pas plus loin dans l'étude de cette variante de Mogbil et Naurois car nous sommes ici intéressé par un critère qui s'affranchisse véritablement des interrupteurs.*

4.3. (In)utilité du témoin d'acyclicité et de connexité

On se pose ensuite la question du témoin d'acyclicité et de connexité fourni par l'interrupteur et on arrive au fait que si un interrupteur vérifie cela, alors pour tous les interrupteurs, on a une composante connexe de plus qu'il n'y a de cycles et on en extrait la version affaiblie avec l'invariant sur $\#ax - \#\otimes$. Cela nous donne le dernier critère. On remarque que connexité et la caractérisation axiome/tenseurs sont évidemment nécessaire et donc il ne nous reste plus qu'à montrer que DepGraph est suffisant pour assurer de la séquentialisation.

Définition 4.14. *Soit G un graphe non-orienté et soient respectivement n, a ses nombres de nœuds et d'arêtes. On notera χ_G la valeur $n - a$, appelée caractéristique du graphe.*

Théorème 4.15. *Soit G un graphe non-orienté acyclique et c_G son nombre de composantes connexes. On a l'égalité suivante : $\chi_G = c_G$.*

Démonstration. Voir en annexe. □

Proposition 4.16. *Pour tout graphe de correction G d'une structure correcte, on a $\chi_G = 1$.*

Démonstration. Immédiat par connexité et acyclicité. □

Proposition 4.17. *Tout graphe de correction G d'une structure R de preuve vérifie : $\chi_G = \#ax - \#\otimes$. On notera χ_R ce nombre, la caractéristique de la structure.*

Démonstration. On fait un simple dénombrement dont le résultat découle immédiatement :
- $n = \#ax + \#\otimes + \#\parr + \#concl$;
- Pour dénombrer les arêtes, on remarque que chaque arête est arête entrante d'exactement un nœud du graphe de correction, ce qui nous donne : $a = 0 \times \#ax + 2 \times \#\otimes + \#\parr + \#concl$.

□

La conjonction des deux propositions précédentes nous indique donc que pour qu'une structure de preuve soit séquentialisable il faut que la différence du nombre de liens axiome et du nombre de liens tenseur soit égale à un ($\#ax - \#\otimes = 1$).

Nous avons maintenant les outils pour énoncer notre nouveau critère, *DepGraph* :

Définition 4.18. ***Critère de graphe de dépendance.*** *Une structure de preuve G vérifie le critère de graphe de dépendance (D-correcte) si :*
- *$D(G)$ est SDAG ;*
- *G est connexe ;*
- *La caractéristique (définie dans la proposition 4.17) de G vaut 1.*

4.4. DepGraph est un critère de correction

Théorème 4.19. *Si une structure de preuve est D-correcte alors elle est un réseau de preuve.*

Pour montrer qu'une structure de preuve D-correcte est un réseau de preuve, nous allons montrer qu'elle vérifie le critère de contractilité (ie. son graphe apparié est contractile). Pour ce faire, nous avons besoin des lemmes suivants :

Lemme 4.20. *La connexité est préservée par les étapes de contractilité.*

Lemme 4.21. *La quantité $n - a_{nap} - a_{ap}$ (où n est le nombre de nœuds du graphe, a_{nap} est son nombre d'arêtes non-appariées et a_{ap} est son nombre de paires d'arêtes appariées) est conservée par les étapes de contractilité.*

Notation. *On définit un s-chemin dans un graphe apparié G comme étant un chemin élémentaire qui ne contient pas deux arêtes appariées. Si p est un s-chemin, on le note aussi $p\{a_1, \ldots, a_n\}$ pour indiquer les arêtes* appariées *par lesquelles il passe.*

On définit les prémisses d'un nœud apparié n via les arêtes a, a' comme étant les nœuds reliés à n par ces arêtes appariées.

La notion de graphe de dépendance est aussi transportée pour les graphes appariés : les nœuds du graphe de dépendance $D(G)$ d'un graphe apparié G sont les nœuds appariés de G et un nœud supplémentaire s. De plus, si x est un nœud apparié,
- Il y a une arête $(s \to x)$ dans $D(G)$ s'il existe un s-chemin $p\emptyset$ dans G entre les deux nœuds prémisses de x.
- Soit y un autre nœud apparié. Il y a une arête $(y \to x)$ s'il existe un s-chemin dans G qui contient l'une des arêtes appariées de y.

Lemme 4.22. *Soit G un graphe apparié et G' tel que $G \to^\star G'$. Soient $\{b_1, \dots, b_m\}$ les arêtes appariées qui ont été contractées. Soient x et y deux nœuds dans G. Si dans G, x et y sont reliées par un s-chemin $p\{a_1, \dots, a_n\}$, alors :*

- *soit les nœuds x et y ont été fusionnés*
- *sinon il existe un s-chemin $q\{a_1, \dots, a_n\} \setminus \{b_1, \dots, b_m\}$ qui relie x et y dans G'.*

Réciproquement, tout s-chemin de G' provient de la contraction d'un s-chemin de G.

Démonstration. On montre le résultat pour $\to$ et on l'étend ensuite à $\to^\star$. Le raisonnement s'effectue en considérant les différentes configurations possibles pour la contraction des arêtes de G et ne pose pas de problème. □

Retour à la preuve du théorème :
Démonstration. Soit R une structure de preuve et $G = C(R)$ son graphe apparié. Nous allons construire une suite de contractions $c_1, \dots, c_n$ telles que, si on pose : $G := G_0 \to^\star_{c_1} G_1 \cdots \to^\star_{c_n} G_n$, on a :

- Le graphe de dépendance de G_{i+1} est le graphe de dépendance de G_i, où un nœud n directement relié à la source a été enlevé et où les nœuds dont le seul parent est n sont reliés à la source par une arête. On montre facilement que le caractère SDAG du graphe de dépendance est invariant par cette transformation.
- Le graphe de dépendance G_n contient l'unique nœuds s, G_n ne contient donc aucun nœud apparié.

Supposons qu'on a construit le i-ème graphe G_i. Soit $D(G_i)$ son graphe de dépendance. Soit n un nœud directement lié à la source. Il existe un chemin $p\emptyset$ qui relie les deux nœuds prémisses de n. En contractant toutes les arêtes de $p\emptyset$ (par l'opération r_2), les deux nœuds prémisses de n se retrouvent fusionnés, on peut alors appliquer r_1 sur les arêtes appariées de n. On note cette séquence de contractions c_{i+1}, et on note G_{i+1} le graphe obtenu à partir de G_i en appliquant c_{i+1} . Analysons l'effet de cette suite de contractions sur les s-chemins de G_i : comme les seules arêtes appariées qui ont été contractées sont a_1 et a_2 qui sont les arêtes appariées de n, par le lemme 4.22, pour tout nœud apparié x, si $p\{E\}$ est un s-chemin qui relie les deux nœuds prémisses de x, alors il existe un s-chemin $q\{E\} \setminus \{a_1, a_2\}$.

Il y a deux cas possibles :

- Si x avait n comme unique parent dans $D(G_i)$, cela signifie que tous les s-chemins qui relient les prémisses de x dans G_i (on sait qu'il en existe au moins un) sont de la forme $p\{a_1\}$ ou $p\{a_2\}$. Ces chemins deviennent des s-chemins $q\emptyset$ dans G_{i+1}. x est alors relié à la source dans $D(G_{i+1})$.
- Si x avait plusieurs parents $\{n_1, \dots, n_k\}$ dans $D(G_i)$, alors ses parents deviennent $\{n_1, \dots, n_k\} \setminus \{n\}$ dans $D(G_{i+1})$.

$D(G_{i+1})$ contient les mêmes nœuds que $D(G_i)$ sauf n et ses arêtes sont les mêmes sauf pour les nœuds qui avaient n comme seul parent et qui sont maintenant reliées à la source.

Comme le graphe de dépendance qu'on obtient à chaque étape est encore SDAG, il existe toujours un nœud directement relié à la source, et de ce fait on peut renouveler cette procédure jusqu'à épuisement des nœuds appariés. On obtient un graphe G_n qui ne contient pas de nœuds appariés. Le graphe G' obtenu en appliquant toutes les contractions r_2 possibles est bien défini.
G' est connexe (la contractilité conserve la connexité par le lemme 4.20) et il ne peut donc contenir qu'un nœud (sinon on pourrait appliquer r_2).
Par le lemme 4.21, la quantité $n - a_{nap} - a_{ap}$ est préservée par les étapes de contraction. Comme elle vaut 1 pour le graphe D-correct G, elle vaut aussi 1 pour le graphe G'. G' ne contient pas de nœuds appariés, on a donc $a_{nap} = O$, et il contient qu'un unique nœud, donc $a_{ap} = 0$. G' est donc le graphe réduit à un seul nœud : G est bien contractile. □

4.5. Comparaison entre les deux notions de graphe de dépendance

L'exemple de la figure 3 nous a montré que les graphes de dépendance à la Mogbil-Naurois dépendent des choix d'interrupteur. On va ici montrer que, pour les réseaux de preuve, on a presque

cette invariance. Plus précisément, les clôtures transitives des graphes des différents interrupteurs (c'est-à-dire l'ordre associé au graphe de dépendance) sont toutes égales et elles sont égales à la clôture transitive du graphe de dépendance que nous avons introduit.

Notations. *Si $\mathcal{S}$ est un interrupteur d'une structure de preuve R et a un lien $\parr$ de cette structure, on note $\mathcal{S}_a$ l'interrupteur $\mathcal{S}$ dans lequel on a inversé le positionnement de a.*

Étant donné un graphe D, on notera D^* sa clôture transitive.

Lemme 4.23. *Soient z et a deux liens $\parr$ d'une structure DR-correcte R et un interrupteur $\mathcal{S}$.*
- *si $(z \to a) \in D(\mathcal{S}, R)$, alors $(z \to a) \in D(\mathcal{S}_a, R)$*
- *si $(a \to z) \in D(\mathcal{S}, R)$, alors $(a \to z) \in D(\mathcal{S}_a, R)$*

Démonstration. Le premier point découle du fait que le chemin élémentaire entre les prémisses de a n'est pas affecté par l'inversion de a.

Pour le second point : si ce n'était pas le cas, on aurait un chemin élémentaire entre les prémisses Z_g et Z_d de z qui ne passe pas par a dans $\mathcal{S}_a(R)$ qui serait également présent dans $\mathcal{S}(R)$, mais par hypothèse on a dans $\mathcal{S}(R)$ un chemin élémentaire entre Z_g et Z_d qui passe par a, ce qui donnerait deux chemins *élémentaires* différents entre Z_g et Z_d dans $\mathcal{S}(R)$, en contradiction avec l'acyclicité. □

Théorème 4.24. *Soit R une structure de preuve DR-correcte. Pour tous $\mathcal{S}, \mathcal{S}'$ interrupteurs de cette structure, on a*

$$D(\mathcal{S}, R)^* = D(\mathcal{S}', R)^*$$

Remarque. *La démonstration utilise constamment le fait que dans un graphe connexe et acyclique, il existe toujours un* unique *chemin élémentaire entre deux nœuds. La preuve ne fonctionnerait pas avec une structure qui n'est pas DR-correcte.*

Démonstration. On montre en fait que pour tout interrupteur $\mathcal{S}$ et pour tout lien $\parr$ a de la structure, on a $D(\mathcal{S}, R) \subseteq D(\mathcal{S}_a, R)^*$. Par symétrie et comme $\mathcal{S}' = \mathcal{S}_{a_1 \dots a_n}$ pour une certaine suite $a_1 \dots a_n$ de liens $\parr$, cela est suffisant.

Soient donc R une structure de preuve DR-correcte, un interrupteur $\mathcal{S}$ de R, x et a deux liens $\parr$ de R. Soit également y tel que $(y \to x) \in D(\mathcal{S}, R)$. On note X_g et X_d les deux prémisses de x.

Si y est s, la source de $D(\mathcal{S}, R)$, alors $(a \to x) \notin D(\mathcal{S}, R)$ par définition. Le chemin élémentaire entre X_g et X_d n'est pas affecté par l'inversion de a et ne passe toujours par aucun lien $\parr$, donc $(s \to x) \in D(\mathcal{S}_a, R)$.

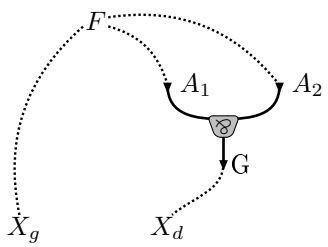
On suppose donc que $y \neq s$, c'est à dire que y est un lien $\parr$ du réseau.

Si $(a \to x) \notin D(\mathcal{S}, R)$, le chemin élémentaire entre X_g et X_d n'est pas affecté par l'inversion de a et passe toujours par y, donc $(y \to x) \in D(\mathcal{S}_a, R)$.

Si $(a \to x) \in D(\mathcal{S}, R)$ et $(y \to a) \in D(\mathcal{S}, R)$, le lemme 4.23 implique que $(y \to a) \in D(\mathcal{S}_a, R)$ et $(a \to x) \in D(\mathcal{S}_a, R)$, et donc $(y \to x) \in D(\mathcal{S}_a, R)^*$.

Finalement, si $(a \to x) \in D(\mathcal{S}, R)$ et $(y \to a) \notin D(\mathcal{S}, R)$, on note
- A_1 la prémisse de a qui appartient au chemin élémentaire entre X_g et X_d dans $\mathcal{S}(R)$
- A_2 la prémisse de a qui appartient au chemin élémentaire entre X_g et X_d dans $\mathcal{S}_a(R)$

On note également F et G les points de branchements de ces deux chemins (l'un des deux, disons G, étant nécessairement la conclusion de a), comme décrit dans le schéma suivant :



(où les lignes pointillées représentent des chemins élémentaires présents à la fois dans $\mathcal{S}(R)$ et $\mathcal{S}_a(R)$)

Comme par hypothèse $(y \to a) \notin D(\mathcal{S}, R)$, y ne peut pas être sur le chemin élémentaire reliant F à A_1, ni sur celui reliant F à A_2 dans $\mathcal{S}(R)$. De plus $(y \to x) \in D(\mathcal{S}, R)$, donc y doit être soit sur le chemin élémentaire reliant X_g à X_d dans $\mathcal{S}(R)$. y doit donc être sur le chemin élémentaire reliant F à X_g ou sur celui reliant G à X_d.

Ces deux chemins étant toujours présents dans $\mathcal{S}_a(R)$, on a bien $(y \to x) \in D(\mathcal{S}_a, R)$. □

Il ne nous reste plus qu'à montrer la dernière partie du résultat :

Théorème 4.25. *Soit R un réseau et soit $\mathcal{S}$ un interrupteur pour R. On a $D(R)^\star = D(\mathcal{S}, R)^\star$.*

Démonstration. En effet, étant donné un interrupteur $\mathcal{S}_0$, on a grace au théorème 4.24

$$D(\mathcal{S}_0, R) \subseteq D(R) = \bigcup_{\mathcal{S} \text{ interrupteur de } R} D(\mathcal{S}, R) \subseteq \bigcup_{\mathcal{S} \text{ interrupteur de } R} D(\mathcal{S}, R)^\star = D(\mathcal{S}_0, R)^\star$$

On en déduit $D(\mathcal{S}_0, R)^\star \subseteq D(R)^\star = D(\mathcal{S}, R)^{\star\star} = D(\mathcal{S}_0, R)^\star$ ce qui donne l'égalité souhaitée. □

5. Conclusion

Comparaison au critère original Le critère DepGraph défini dans la section 4.2 reprend la notion de graphe de dépendance introduite dans [8] et la rend indépendante de la notion d'interrupteur. Le choix de le formuler dans un cadre très restreint (MLL) est fait pour favoriser l'exposition, mais ces résultats s'étendent à MELL de la manière habituelle.

La preuve du théorème de séquentialisation met en évidence que la condition (SDAG) sur le graphe de dépendance est d'une nature différente des deux autres conditions de la définition 4.18. En passant par la contractilité, on a pu voir que la première condition (SDAG) assure l'absence de *deadlocks* causés par des arêtes appariées, tandis que les deux autres (connexité et caractéristique égale à 1) forcent la forme normale à être réduite à un point.

En particulier, l'ordre induit par le graphe de dépendance d'une structure correcte donne une information sur les liens de causalité entre connecteurs ⅋. Le théorème 4.11 montre que toute séquentialisation d'un réseau correct devra *a minima* respecter cet ordre.

Perspectives Une des directions futures de notre travail serait d'explorer plus en détails les rapports entre graphe de dépendance et causalité dans les preuves. Il serait par exemple intéressant de prouver une "réciproque" du théorème 4.11 : si toutes les séquentialisations d'un réseau voient un lien ⅋ introduit avant un autre, cela devrait être visible sur le graphe de dépendance.

Les théorèmes 4.24 et 4.25 sont encourageants : ils montrent que derrière le graphe de dépendance se cache un invariant des séquentialisations d'un réseau, l'ordre induit par le graphe de dépendance.

Références

[1] Jean-Marc Andreoli : *Proposition pour une synthèse des paradigmes de la programmation logique et de la programmation par objets.* Thèse de doctorat, Université Paris VI, juin 1990.

[2] Gérard Berry : Stable models of typed lambda-calculi. pages 72–89, 1978.

[3] Pierre-Louis Curien : Introduction to linear logic and ludics, part ii, 2006.

[4] Pierre-Louis Curien et Guillaume Munch-Maccagnoni : The duality of computation under focus. *In IFIP TCS*, volume 323, pages 165–181. Springer, 2010.

[5] Vincent Danos : *Une application de la logique linéaire à l'étude des processus de normalisation (principalement du λ-calcul).* Thèse de doctorat, Université Denis Diderot, Paris 7, 1990.

[6] Vincent Danos, Jean-Baptiste Joinet et Harold Schellinx : A new deconstructive logic : Linear logic. 62(3):755–807, 1997.

[7] Vincent Danos et Laurent Regnier : The structure of multiplicatives. 28:181–203, 1989.

[8] Paulin Jacobé de Naurois et Virgile Mogbil : Correctness of linear logic proof structures is NL-complete. *Theor. Comput. Sci.*, 412(20):1941–1957, 2011.

[9] Amina Doumane : *Géométrie de l'Interation VI.* Mémoire de master 2, Université Paris Diderot, septembre 2013.

[10] Thomas Ehrhard et Laurent Regnier : Differential interaction nets. *Theor. Comput. Sci.*, 364(2):166–195, 2006.

[11] Jean-Yves Girard : *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de doctorat d'etat, Université Paris VII, 1972.

[12] Jean-Yves Girard : The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[13] Jean-Yves Girard : Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987.

[14] Jean-Yves Girard : A new constructive logic : classical logic. 1(3):255–296, 1991.

[15] Jean-Yves Girard : Proof-nets : the parallel syntax for proof-theory. *In* Aldo Ursini et Paolo Agliano, éditeurs : *Logic and Algebra*, volume 180 de *Lecture Notes In Pure and Applied Mathematics*, pages 97–124, New York, 1996. Marcel Dekker.

[16] Jean-Yves Girard : Light linear logic. *Information and Computation*, 143(2):175–204, juin 1998.

[17] Jean-Yves Girard : *Le Point Aveugle : Cours de logique. Tome 1, Vers la perfection ; Tome 2, Vers l'imperfection.* Hermann, 2006.

[18] Georges Gonthier, Martin Abadi et Jean-Jacques Lévy : The geometry of optimal lambda reduction. *In Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 15–26, 1992.

[19] Stefano Guerrini : Correctness of multiplicative proof nets is linear. *In LICS*, pages 454–463. IEEE Computer Society, 1999.

[20] Stefano Guerrini : A linear algorithm for MLL proof net correctness and sequentialization. *Theor. Comput. Sci.*, 412(20):1958–1978, 2011.

[21] William A. Howard : The formulae-as-type notion of construction, 1969. *In* J. P. Seldin et R. Hindley, éditeurs : *To H. B. Curry : Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980.

[22] Yves Lafont : Interaction nets. *In POPL90*, pages 95–108, San Francisco, California, 1990. ACM Press.

[23] Yves Lafont : Soft linear logic and polynomial time. *TCS*, 318(1–2):163–180, juin 2004.

[24] Olivier Laurent : *Étude de la polarisation en logique.* Thèse de doctorat, mars 2002.

[25] Dale Miller : Overview of linear logic programming. *In* Thomas Ehrhard, Jean-Yves Girard, Paul Ruet et Phil Scott, éditeurs : *Linear Logic in Computer Science*, volume 316 de *London Mathematical Society Lecture Note*, pages 119–150. Cambridge University Press, 2004.

[26] Misao Nagayama et Mitsuhiro Okada : A new correctness criterion for the proof nets of non-commutative multiplicative linear logics. *J. Symb. Log.*, 66(4):1524–1542, 2001.

[27] John C. Reynolds : Towards a theory of type structure. *In Symposium on Programming*, volume 19 de *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[28] Noam Zeilberger : Focusing and higher-order abstract syntax. *In POPL*, pages 359–369. ACM, 2008.

A. Preuve du théorème 4.11

Théorème A.1. *Soit π une preuve MLL et R le réseau de preuve correspondant. On a :*

$$D(R) \subseteq O(\pi)$$

Démonstration. Par induction sur π.

- Supposons que la preuve π de $\vdash \Gamma, A \parr B$ a été obtenue a partir de la preuve ν de $\vdash \Gamma, A, B$. On notera R_π et R_ν les réseaux de preuves respectifs de π et ν.
  Comme $A \parr B$ est en dessous de toutes les formules introduites par une règle $\parr$ dans π, et comme l'ordre sur toutes les autres formules est le même dans π et dans ν, le graphe $O(\pi)$ est constitué du graphe $O(\pi)$, du nœud $A \parr B$ et d'arêtes partant des nœuds de $O(\pi)$ et arrivant sur $A \parr B$.

  D'autre part, $D(R_\pi)$ est constitué du graphe $D(R_\nu)$, d'un nœud $A \parr B$ et de certaines arêtes partant des nœuds de $D(R_\nu)$ et arrivant sur $A \parr B$. En effet, on ne peut pas créer de nouvelles arêtes entre les nœuds de $D(R_\nu)$, car cela signifierait qu'il existe un chemin élémentaire entre les nœuds $\parr$ correspondants dans π qui n'existait pas dans ν. Un tel chemin passerait forcément par les prémisses gauche et droite consécutivement du nœud $A \parr B$, ce qui est absurde. De plus, il ne peut exister une arête de $A \parr B$ vers l'un des nœuds de ν car ici encore le chemin élémentaire correspondant passerait consécutivement par les deux prémisses du nœuds $A \parr B$ ce qui est absurde. Or, par hypothèse d'induction, on a $D(R_\nu) \subseteq O(\nu)$. Par ce qui précède, on conclut qu'on a aussi $D(R_\pi) \subseteq O(\pi)$.
- Supposons que la preuve π de $\vdash \Gamma, A \otimes B$ a été obtenue a partir de la preuve π_1 de $\vdash \Gamma, A$ et de la preuve π_2 de $\vdash \Delta, B$. On notera R_π, R_{π_1} et R_{π_2} les réseaux de preuves respectifs de π, π_1 et π_2.
  Il est facile de voir que $O(\pi)$ est l'union de $O(\pi_1)$ et $O(\pi_2)$.

  D'autre part, $D(R_\pi)$ est l'union de $D(R_{\pi_1})$ et $D(R_{\pi_2})$. En effet, s'il y a un chemin élémentaire entre deux nœuds $\parr$ appartenant respectivement à R_{π_1} et R_{π_2}, celui-ci contiendrait forcément un cycle puisqu'il doit passer deux fois par le nœud $A \otimes B$, ce qui est absurde.

  Or, par hypothèse d'induction, on a $D(R_{\pi_1}) \subseteq O(\pi_1)$ et $D(R_{\pi_2}) \subseteq O(\pi_2)$. Par ce qui précède, on conclut qu'on a aussi $D(R_\pi) \subseteq O(\pi)$.

□

B. Preuve du théorème 4.15

Théorème B.1. *Soit G un graphe non-orienté acyclique et c_G son nombre de composantes connexes. On a l'égalité suivante : $\chi_G = c_G$.*

Démonstration. Par induction sur le nombre d'arêtes du graphe.

Si le graphe ne contient aucune arête, on a autant de composantes connexes que de nœuds : $a = 0$ et $n = c$ et l'égalité est vérifiée.

Supposons l'égalité est valide pour les graphes acycliques comportant k arêtes, et soit G un graphe acyclique à $k+1$ arêtes. En retirant une arête de G, on obtient G' qui est acyclique, a une composante connexe de plus et une arête de moins que G : $\chi_G + 1 = \chi_{G'}$ et $c_G + 1 = c_{G'}$. Par induction $\chi_{G'} = c_{G'}$ et donc $\chi_G = c_G$. □

Nécessité faite loi

Pierre-Marie Pédrot[1] & Alexis Saurin[1]

Laboratoire PPS,
CNRS, UMR 7126, Univ Paris Diderot, Sorbonne Paris Cité,
PiR2, INRIA Paris Rocquencourt, F-75205 Paris, France
{pedrot,saurin}@pps.univ-paris-diderot.fr

Résumé

À partir de la réduction linéaire de tête, nous dérivons de manière systématique un calcul en appel par nécessité. L'introduction d'un calcul pour la réduction linéaire de tête, basée sur une analyse fine de la notion de radicaux premiers de Danos et Regnier, nous permet de construire pas à pas un lambda-calcul en appel par nécessité que l'on compare aux calculs présents dans la littérature.

1. Introduction

Évaluation paresseuse. L'évaluation paresseuse a été proposée par Wadsworth dans sa thèse [18] comme une manière de résoudre une tension entre les stratégies en appel par nom et en appel par valeur. En posant comme il est habituel $\Delta \equiv \lambda x.\,(x)\,x$, $I \equiv \lambda y.\,y$ et en notant $\to_{\mathrm{cbn}}$ (resp. $\to_{\mathrm{cbv}}$) la réduction associée à l'appel par nom (resp. par valeur), on constate aisément que l'appel par valeur est susceptible d'effectuer des calculs inutiles, qui peuvent d'ailleurs conduire à une non-terminaison du calcul (on mettra en évidence le radical impliqué dans une réduction en le représentant de manière grisée) :

$$\begin{array}{rcl} t & \equiv & \boxed{(\lambda x.\,I)\,((\Delta)\,\Delta)} \to_{\mathrm{cbn}} I \\ t & \equiv & (\lambda x.\,I)\,\boxed{((\Delta)\,\Delta)} \to_{\mathrm{cbv}} t \to_{\mathrm{cbv}} \dots \end{array}$$

Ici, l'appel par valeur va s'obstiner à réduire $(\Delta)\,\Delta$ sans s'apercevoir que cet argument ne sert à rien.

On constate tout aussi aisément que l'appel par nom est conduit à effectuer des calculs redondants :

$$\begin{array}{rcl} u & = & \boxed{(\Delta)\,((I)\,I)} \to_{\mathrm{cbn}} \boxed{((I)\,I)}\,((I)\,I) \to_{\mathrm{cbn}} \boxed{(I)\,((I)\,I)} \to_{\mathrm{cbn}} \boxed{(I)\,I} \to_{\mathrm{cbn}} I \\ u & = & (\Delta)\boxed{((I)\,I)} \to_{\mathrm{cbv}} \boxed{(\Delta)\,I} \to_{\mathrm{cbn}} \boxed{(I)\,I} \to_{\mathrm{cbn}} I \end{array}$$

L'appel par nom va ici dupliquer le calcul de $(I)\,I$ alors que l'appel par valeur, ayant fait le calcul avant la substitution, va dupliquer une valeur.

L'appel par nécessité consiste à résoudre cette tension en cherchant à n'effectuer un calcul qu'au cas où ce calcul est nécessaire à la progression de l'évaluation et, dans ce cas, à éviter de répéter des calculs inutilement. La solution de Wasdworth consiste à introduire une réduction de graphe qui permet de partager effectivement des sous-termes. L'appel par nécessité est résumé de plusieurs manières dans la littérature. Ainsi, Danvy et al. [10] résume l'appel par nécessité par la formule suivante :

Demand-driven computation & memoization of intermediate results [10]

En ce sens, l'appel par nécessité est une optimisation aussi bien de l'appel par nom que de l'appel par valeur. Il est cependant plus proche de l'appel par nom que de l'appel par valeur car il a la même notion de convergence et en cela l'équivalence observationnelle associée à l'appel par nécessité coïncide avec celle de l'appel par nom.

Le λ-calcul par nécessité d'Ariola et Felleisen. En 1994, deux groupes de chercheurs ont, de manière simultanée et indépendante, proposé des λ-calculs pour l'appel par nécessité, d'un côté Ariola et Felleisen, de l'autre Maraist, Odersky et Wadler. Les deux propositions, soumises à la même conférence, ont donné lieu à une publication conjointe à POPL 1995 [4] mais à deux versions journal distinctes [3, 15], les auteurs faisant des choix de conception différents. Le calcul d'Ariola et Felleisen peut être présenté comme suit :

Définition 1. Le λ-calcul par nécessité d'Ariola et Felleisen est défini par la syntaxe et les règles de réduction suivantes :

Syntaxe

Terme	t, u	$::=$	$x \mid \lambda x.\, t \mid (t)\, u$
Valeur	v	$::=$	$\lambda x.\, t$
Réponses	A	$::=$	$v \mid \lambda x.\, A\, t$
Contexte d'évaluation	E	$::=$	$[\cdot] \mid E\, t \mid (\lambda x.\, E)\, t \mid \lambda x.\, E[x]\, E$

Réductions

(Deref)	$(\lambda x.\, E[x])\, v$	$\to$	$(\lambda x.\, E[v])\, v$
(Lift)	$(\lambda x.\, A)\, t\, u$	$\to$	$(\lambda x.\, (A)\, u)\, t$
(Assoc)	$(\lambda x.\, E[x])\, (\lambda y.\, A)\, t$	$\to$	$(\lambda y.\, (\lambda x.\, E[x])\, A)\, t$

Intuitivement, la syntaxe et les règles précédentes doivent être comprises comme suit :
- C'est la structure des contextes qui est responsable du caractère paresseux de ce calcul : le terme $E[x]$ met en évidence le fait que la variable x est en position nécessaire dans le terme $E[x]$.
- La règle Deref va alors permettre d'aller chercher le contenu d'un argument à condition qu'il soit déjà calculé et que cet argument soit identifié comme nécessaire car il est passé à une variable nécessaire ; dans ce cas, une substitution a lieu, mais linéairement. L'application n'a en effet pas disparu, et seule une occurrence de la variable a été substituée. (E est en effet un contexte à un seul trou.)
- Les deux autres règles Lift et Assoc vont alors autoriser la commutation des contextes d'évaluation afin de permettre à des radicaux β d'apparaître malgré les lieurs qui persistent.

Exemple de réduction dans le calcul d'Ariola et Felleisen.

$$
\begin{array}{lll}
(\Delta)(I)I & \equiv & (\lambda x.\, (x)\, x)\, (\lambda y.\, y)\, I \\
& \to_{\text{Deref}} & (\lambda x.\, (x)\, x)\, (\lambda y.\, I)\, I \\
& \to_{\text{Assoc}} & (\lambda y.\, (\lambda x.\, (x)\, x)\, I)\, I \\
& \to_{\text{Deref}} & (\lambda y.\, (\lambda x.\, (\lambda z.\, z)\, x)\, I)\, I \\
& \to_{\text{Deref}} & (\lambda y.\, (\lambda x.\, (\lambda z.\, z)\, I)\, I)\, I \\
& \to_{\text{Deref}} & (\lambda y.\, (\lambda x.\, (\lambda z.\, I)\, I)\, I)\, I \\
& \to_{gc}^{\star} & I
\end{array}
$$

où gc désigne une réduction éliminant les β-radicaux rendus inutiles par le fait que la variable qu'ils lient n'est pas utilisée dans le corps de l'abstraction. Cette notion de réduction ne fera pas partie de nos réductions mais permet de faciliter la lecture des valeurs ; on utilisera cette notation dans la suite de l'article.

Remarque 1 (Sur l'utilisation d'une syntaxe avec `let`). Il est standard de présenter les calculs en appel par nécessité en étendant le λ-calcul avec une construction `let` qui exprime le partage : $\mathbf{let}\, x = t\, \mathbf{in}\, u$. Dans ce cas, on rajoute une règle pour introduire les `let` :

$$(\beta) \qquad (\lambda x.\, t)\, u \;\to\; \mathbf{let}\, x = u\, \mathbf{in}\, t$$

En ce sens, la construction `let` n'est rien d'autre que le marquage du fait qu'un β-radical a été rencontré.

Même s'il y a des intérêts à travailler avec un `let` explicite, que ce soit du point de vue de l'appel par nécessité ou des liens logiques avec le calcul des séquents [7], nous faisons le choix dans cet article de rester fidèle à la syntaxe du λ-calcul et cela pour une raison fort simple : la réduction linéaire de tête s'exprime très naturellement dans cette syntaxe et le calcul initial d'Ariola et Felleisen y est aussi défini [1]. Par ailleurs, nous allons adopter une approche macroscopique de l'évaluation, compatible avec la réduction linéaire de tête telle que présentée par Danos et Regnier ; dans ce cadre il n'y aurait pas vraiment de sens à vouloir introduire les `let`.

L'objet de cet article étant de mettre en évidence les liens profonds entre réduction linéaire de tête et évaluation paresseuse, il est donc naturel de s'en tenir à la syntaxe du λ-calcul.

Des appels par nécessité ? Le calcul présenté par Maraist, Odersky et Wadler dans leur version finale de 1998 diffère du calcul ci-dessus par plusieurs aspects, mais les deux calculs partagent la même réduction standard.

Dans la mesure où l'appel par nécessité est une optimisation de l'appel par nom visant à résoudre les limitations mentionnées plus haut sur les duplications de calculs, et que l'équivalence observationnelle est identique à celle induite par l'appel par nom, il peut sembler futile de vouloir comparer les différents appels par nécessité et mettre en évidence un appel par nécessité comme étant plus canonique qu'un autre.

Pourtant, dès que l'on souhaite introduire des opérateurs de contrôle en appel par nécessité, on est obligé de changer de perspective : en effet, en ajoutant des opérateurs de contrôle, non seulement on rend observable la différence entre appel par nom et appel par nécessité, mais en outre on peut distinguer plusieurs variantes de l'appel par nécessité [5]. Dans ce contexte, cela fait sens que de vouloir trouver un appel par nécessité canonique.

Contributions et organisation de l'article Dans cet article, nous montrerons que l'on peut retrouver un calcul connu en partant d'une réduction construite via des considérations héritées de la logique linéaire, définie à la section 2 et en le raffinant par étapes successives :

1. Un appel par nom faible généralisé (section 3.1) ;
2. Une mémoïsation par passage de valeurs (section 3.2) ;
3. Un partage de réductions (section 3.3).

Cette méthodologie trouve sa justification dans le fait que le calcul final se trouve être un calcul en appel par nécessité, ce que nous justifierons par la suite. Cela nous conduit donc au slogan :

$$\boxed{\text{Nécessité} = \text{Calcul à la demande} + \text{Mémoïzation} + \text{Partage}} \qquad (\textsc{Slogan})$$

ou, plus précisément, à affirmer que l'appel par nécessité n'est rien d'autre qu'une *(i)* réduction linéaire de tête faible, *(ii)* en appel par valeur *(iii)* effectuant un partage des clôtures.

Pour atteindre ce résultat, nous commencerons par examiner en détail la réduction linéaire de tête ce qui nous conduira à en proposer une formulation nouvelle sous forme de calcul. La reformulation de la réduction linéaire de tête et les transformations qui vont suivre reposent essentiellement sur deux idées :

1. Plus exactement, Ariola et Felleisen considèrent les deux présentations du calcul, les `let` servant essentiellement à montrer la complétude de leur calcul vis-à-vis de l'appel par nom

– se laisser guider par la σ-équivalence (que nous discutons en section 2.2) ;
– mettre en évidence la structure de contexte de clôture qui constitue l'une des notions clés de notre construction.

C'est cette reformulation de la réduction linéaire de tête qui nous met sur la voie des transformations suivantes : on restreindra d'abord la réduction à être une réduction faible. On limitera le calcul de manière à ce qu'il ne substitue que des valeurs. Enfin, on atteindra un calcul en appel par nécessité en ajoutant une règle pour partager les clôtures, il s'agit du calcul de Chang et Felleisen [6] à ceci près qu'on effectue la substitution linéaire avec des abstractions persistantes alors que leur calcul effectue une réduction destructrice. Avant de conclure, on discute des extensions pleinement paresseuses des calculs en appel par nécessité.

2. Réduction linéaire de tête

Dans cette section, nous rappelons la définition de la réduction linéaire de tête, sur laquelle se base notre formulation de l'appel par nécessité.

2.1. Présentation historique

Introduite par Danos et Regnier [8, 9], la réduction linéaire de tête a été formulée à la suite d'observations similaires au sein de différents systèmes calculatoires, comme les réseaux de preuve [11, 16], la machine de Krivine [13] et la sémantique des jeux [12]. Selon eux, la réduction implémentée par ces systèmes en sous-main est la réduction de tête linéaire et non pas la réduction de tête habituelle. En effet, ces deux réductions sont observationellement équivalentes dans le λ-calcul pur, et nécessitent la présence d'effets pour être distinguées. Le même phénomène est à l'œuvre dans la distinction entre appel par nom et appel par nécessité. Un effet collatéral de cette apparente identité est l'inexistence quasi-totale de la notion de réduction de tête linéaire dans la littérature.

Afin de pallier ce manque, et dans un souci d'explication, nous redéfinissons ici les notions afférentes. Dans la suite de cette section, nous travaillerons implicitement à α-conversion près et avec les conventions de Barendregt pour éviter la capture inopinée de variables.

Définition 2 (Épine dorsale, variable hoc)**.** Soit t un λ-terme. L'épine dorsale de t est un ensemble $\restriction t$ de λ-termes défini inductivement par :

$$\frac{}{t \in \restriction t} \qquad \frac{r \in \restriction t}{r \in \restriction t\, u} \qquad \frac{r \in \restriction t}{r \in \restriction \lambda x.\, t}$$

De plus, pour tout terme t, $\restriction t$ contient par construction exactement une variable dénotée $\mathrm{hoc}\,(t)$ [2].

L'épine dorsale d'un terme t n'est rien d'autre que l'ensemble des sous-termes gauches de t, et $\mathrm{hoc}(t)$ en est la variable la plus à gauche.

Définition 3 (Lambdas de tête, radicaux premiers)**.** Soit t un λ-terme. On définit mutuellement les lambdas de tête $\lambda_h(t)$ et les radicaux premiers $p(t)$ de t par induction sur t comme suit :

$$\begin{array}{rcl@{\qquad}rcl}
\lambda_h(x) & \equiv & \varepsilon & p(x) & \equiv & \emptyset \\
\lambda_h(\lambda x.\, t) & \equiv & x :: \lambda_h(t) & p(\lambda x.\, t) & \equiv & p(t) \\
\lambda_h(t\, u) & \equiv & \left\{ \begin{array}{ll} \varepsilon & \text{si } \lambda_h(t) = \varepsilon \\ \ell & \text{si } \lambda_h(t) = x :: \ell \end{array} \right. & p(t\, u) & \equiv & \left\{ \begin{array}{ll} p(t) & \text{si } \lambda_h(t) = \varepsilon \\ p(t) \cup \{x \leftarrow u\} & \text{si } \lambda_h(t) = x :: \ell \end{array} \right.
\end{array}$$

2. Pour *head occurrence*, et par assimilation plaisante avec le *hoc* latin.

Remarque 2. On peut voir les lambdas de tête et les radicaux premiers d'une manière alternative, en considérant des blocs d'applications. On a en effet les égalités suivantes :

$$\lambda_h((\lambda x.\, t)\, u\, \vec{r}) = \lambda_h(t\, \vec{r})$$
$$p((\lambda x.\, t)\, u\, \vec{r}) = \{x \leftarrow u\} \cup p(t\, \vec{r})$$

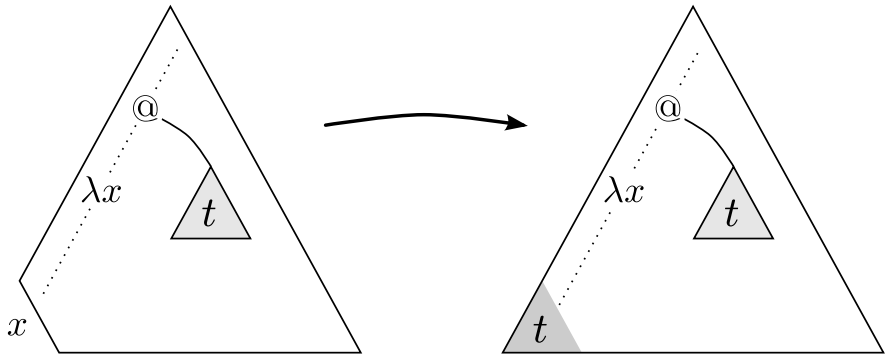
Les lambdas de tête sont les lambdas de l'épine dorsale qui ne recevront pas d'arguments au cours d'une réduction de tête.

Muni de ces notions qui sont illustrées ci-dessous, on peut maintenant définir formellement la réduction linéaire de tête.

Définition 4 (Réduction linéaire de tête)**.** Soit u un λ-terme, soit $x := \mathrm{hoc}\,(u)$. On dit que u se réduit linéairement de tête vers r, noté $u \to_{lh} r$ si :

1. il existe un terme t tel que $\{x \leftarrow t\} \in p(u)$;
2. r est le terme u dans lequel $\mathrm{hoc}\,(u)$, et *uniquement* cette occurrence, a été substituée par t.

Graphiquement :



Remarque 3. La réduction linéaire de tête ne substitue qu'une occurrence d'une variable à la fois, et ne réduit jamais un nœud application, pas plus qu'elle ne diminue le nombre de radicaux premiers : les termes ne font que grossir, d'où le terme de « linéaire » (au sens de *substitution* linéaire).

Voici un exemple de réduction de tête linéaire :

$$\begin{array}{lll}
\Delta\,(I\,I) & \equiv & (\lambda x.\, x\, x)\, ((\lambda y.\, y)\, (\lambda z.\, z)) \\
& \to_{lh} & (\lambda x.\, ((\lambda y_0.\, y_0)\, (\lambda z_0.\, z_0))\, x)\, ((\lambda y.\, y)\, (\lambda z.\, z)) \\
& \to_{lh} & (\lambda x.\, (\lambda y_0.\, \lambda z_1.\, z_1)\, (\lambda z_0.\, z_0)\, x)\, ((\lambda y.\, y)\, (\lambda z.\, z)) \qquad (\star) \\
& \to_{lh} & (\lambda x.\, (\lambda y_0.\, \lambda z_1.\, x)\, (\lambda z_0.\, z_0)\, x)\, ((\lambda y.\, y)\, (\lambda z.\, z)) \\
& \to_{lh} & (\lambda x.\, (\lambda y_0.\, \lambda z_1.\, (\lambda y_2.\, y_2)\, (\lambda z_2.\, z_2))\, (\lambda z_0.\, z_0)\, x)\, ((\lambda y.\, y)\, (\lambda z.\, z)) \\
& \to_{lh} & (\lambda x.\, (\lambda y_0.\, \lambda z_1.\, (\lambda y_2.\, (\lambda z_3.\, z_3))\, (\lambda z_2.\, z_2))\, (\lambda z_0.\, z_0)\, x)\, ((\lambda y.\, y)\, (\lambda z.\, z)) \\
& \to_{gc} & I
\end{array}$$

2.2. Réduction à σ-équivalence près

On remarquera avec profit que la réduction linéaire de tête réduit des termes qui ne sont pas encore des radicaux pour β_h, c'est-à-dire que lh peut aller chercher l'argument d'un lieur qui ne lui est pas directement appliqué. La troisième réduction de l'exemple de la section précédente, notée $(\star)$, en est une illustration. Le lieur λz_1 y saute le radical premier $\{y_0 \leftarrow \lambda z_0.\, z_0\}$ pour aller chercher son argument x. Cette réduction n'aurait pas été possible avec la réduction de tête, β_h.

Ce comportement peut être formalisé plus avant, au travers d'une réécriture à équivalence près, introduite aussi par Regnier [17, 16].

Définition 5 (σ-équivalence)**.** Deux termes t et u sont σ-équivalents, noté $t \cong_\sigma u$, s'ils sont en relation par la clôture des deux générateurs suivants :

$$\begin{array}{rcl} (\lambda x_1.\,(\lambda x_2.\,u)\,t_2)\,t_1 & \cong_\sigma & (\lambda x_2.\,(\lambda x_1.\,u)\,t_1)\,t_2 \\ (\lambda x.\,\lambda y.\,t)\,u & \cong_\sigma & \lambda y.\,(\lambda x.\,t)\,u \end{array}$$

avec les conditions de fraîcheur sur les variables attendues.

La σ-équivalence capture l'intuition que la réduction d'un terme peut faire apparaître des radicaux β qui étaient bloqués par d'autres radicaux essentiellement transparents.

Proposition 1. *Si $t \cong_\sigma u$, alors $p(t) = p(u)$.*

La proposition qui suit souligne le fort lien de parenté qui unit la réduction linéaire de tête avec la σ-équivalence. On rappelle qu'un contexte gauche E est défini par la grammaire inductive suivante :

$$E ::= [\cdot] \mid E\,u \mid \lambda x.\,E$$

Proposition 2. *Si $t \to_{lh} r$ alors il existe deux contextes gauches E_1 et E_2 tels que $t \cong_\sigma E_1[(\lambda x.\,E_2[x])\,u]$ et $r \cong_\sigma E_1[(\lambda x.\,E_2[u])\,u]$.*

En fait, le résultat peut encore être raffiné : la relation $\cong_\sigma$ étant réversible, on peut retrouver r en appliquant à $E_1[(\lambda x.\,E_2[u])\,u]$ les étapes de σ-équivalence inverses de celles appliquées à t pour trouver $E_1[(\lambda x.\,E_2[x])\,u]$. Nous ne détaillerons pas ici précisément cette opération, qui serait peu éclairante et fastidieuse au regard de la présentation qui suit.

2.3. Contextes de clôture

On veut désormais donner un statut de première classe à cette réduction « à σ-équivalence près » en évitant de recourir aux artifices oratoires évoqués juste au-dessus. Pour cela, nous introduisons un contexte de réduction d'un nouveau genre, appelé *contexte de clôture*.

Définition 6 (Contexte de clôture)**.** Les contextes de clôture sont générés par la grammaire inductive suivante :

$$\mathcal{C} ::= [\cdot] \mid (\mathcal{C}_1[\lambda x.\,\mathcal{C}_2])\,t$$

Aussi hétérodoxes que ces contextes puissent paraître, ils possèdent toutes les propriétés requises pour avoir un bon comportement algébrique, à savoir la compositionnalité et la factorisation.

Proposition 3 (Composition)**.** *Si $\mathcal{C}_1$ et $\mathcal{C}_2$ sont deux contextes de clôture, alors $\mathcal{C}_2;\mathcal{C}_1 \equiv \mathcal{C}_1[\mathcal{C}_2]$ est aussi un contexte de clôture.*

Proposition 4 (Factorisation)**.** *Tout terme t peut être uniquement décomposé en un contexte de clôture maximal (au sens habituel de la composition) et un sous-terme t_0.*

Clôtures	c	$::=$	(t, σ)
Environnements	σ	$::=$	$\emptyset \mid \sigma + (x := c)$
Piles	π	$::=$	$\varepsilon \mid c \cdot \pi$
Processus	p	$::=$	$\langle c \mid \pi \rangle$

Push	$\langle (t\,u, \sigma) \mid \pi \rangle$	$\to$	$\langle (t, \sigma) \mid (u, \sigma) \cdot \pi \rangle$
Pop	$\langle (\lambda x.\, t, \sigma) \mid c \cdot \pi \rangle$	$\to$	$\langle (t, \sigma + (x := c)) \mid \pi \rangle$
Grab	$\langle (x, \sigma + (x := c)) \mid \pi \rangle$	$\to$	$\langle c \mid \pi \rangle$
Garbage	$\langle (x, \sigma + (y := c)) \mid \pi \rangle$	$\to$	$\langle (x, \sigma) \mid \pi \rangle$

Figure 1 – Machine de Krivine (avec clôtures)

Mieux encore, ces contextes capturent précisément la notion de radicaux premiers.

Proposition 5. *Soit t un terme. Alors $\{x \leftarrow u\} \in p(t)$ si et seulement si il existe un contexte gauche E, un contexte de clôture $\mathcal{C}$ et un terme t_0 tels que :*

$$t = E[\mathcal{C}[\lambda x.\, t_0]\, u]$$

Moralement, ces contextes sont transparents pour une certaine notion de réduction de tête. On peut en effet voir le contexte $(\mathcal{C}[\lambda x.\, [\cdot]])\, t$ comme un contexte qui ne ferait que d'ajouter la liaison $(x := t)$, et récursivement les liaisons contenues dans $\mathcal{C}$, dans l'environnement.

On peut rendre formelle cette intuition en recourant à la machine de Krivine. On rappelle à la figure 1 la définition de la machine abstraite de Krivine avec clôture (KAM). Comme le montre le résultat suivant, la KAM implémente le calcul des contextes de clôture à l'aide des transitions Push et Pop.

Proposition 6. *Soient t un terme, σ un environnement et π une pile. Pour tout contexte de clôture $\mathcal{C}$, on a :*

$$\langle (\mathcal{C}[t], \sigma) \mid \pi \rangle \longrightarrow^*_{\text{Push, Pop}} \langle (t, \sigma + [\mathcal{C}]_\sigma) \mid \pi \rangle$$

où $[\mathcal{C}]_\sigma$ est définie par induction sur $\mathcal{C}$ comme suit :
- $[[\cdot]]_\sigma \equiv \emptyset$
- $[(\mathcal{C}_1[\lambda x.\, \mathcal{C}_2])\, t]_\sigma \equiv [\mathcal{C}_1]_\sigma + (x := (t, \sigma)) + [\mathcal{C}_2]_{\sigma + [\mathcal{C}_1]_\sigma + (x := (t, \sigma))}$

Réciproquement, pour tous t_0 et σ_0 tels que

$$\langle (t, \sigma) \mid \pi \rangle \longrightarrow^*_{\text{Push, Pop}} \langle (t_0, \sigma_0) \mid \pi \rangle$$

alors il existe $\mathcal{C}_0$ tel que $t = \mathcal{C}_0[t_0]$, où $\mathcal{C}_0$ est défini par induction sur σ_0.

La machine de Krivine fait plus que de calculer les contextes de clôture, grâce aux règles d'accès aux variables Grab et Garbage, qui lui permettent d'implémenter la réduction de tête faible. Mais la proposition précédente souligne bien la similarité de la réduction de tête avec sa cousine linéaire.

En particulier, si l'on désirait fournir une machine abstraite construisant les contextes de clôture, on pourrait simplement adapter la KAM.

2.4. Le calcul λ_{lh}

La présentation historique de la réduction linéaire de tête présente ce défaut qu'est sa définition au travers de structures ad-hoc et peu élégantes, nuisant potentiellement à sa compréhension et à sa manipulation.

En nous appuyant sur le fait que les contextes de clôture capturent les radicaux premiers, on donnera une définition alternative et plus conventionnelle à la réduction linéaire de tête, à base de contextes de réduction. On appellera ce calcul λ_{lh}.

Définition 7 (Calcul λ_{lh})**.** Le calcul λ_{lh} est défini par l'unique règle de réduction

$$E_1[(\mathcal{C}[\lambda x.\, E_2[x]])\, u] \to_{\lambda_{lh}} E_1[(\mathcal{C}[\lambda x.\, E_2[u]])\, u]$$

où E_1, E_2 sont des contextes gauches, $\mathcal{C}$ un contexte de clôture, t et u des termes, et avec les conventions habituelles pour empêcher la capture de variables dans u.

On peut remarquer au passage que cette règle ressemble précisément au calcul du hoc d'un terme et de l'argument correspondant. Dans la règle donnée, x est bien le hoc du terme et, de par le fait qu'on s'autorise de raisonner à contexte de clôture près, on tire de la proposition 5 que $\{x \leftarrow u\}$ est un radical premier. D'où le résultat attendu suivant :

Proposition 7. *Le calcul λ_{lh} capture la réduction linéaire de tête, i.e.*

$$t \to_{\lambda_{lh}} r \qquad \textit{ssi} \qquad t \to_{lh} r$$

On a donc enfin donné une présentation classique à la réduction linéaire de tête.

3. Vers l'appel par nécessité

Dans la section précédente, on a exprimé la réduction linéaire de tête comme un calcul, λ_{lh}. On va maintenant montrer comment dériver, en trois étapes successives, un λ-calcul en appel par nécessité à partir de ce calcul. On va successivement : *(i)* se limiter à une réduction faible, *(ii)* restreindre la substitution à ne passer que des valeurs et *(iii)* autoriser du partage de contextes de clôture. Au terme de ces trois étapes, on aura obtenu un calcul en appel par nécessité. Plus spécifiquement, on comparera notre calcul à une présentation de l'appel par nécessité introduite récemment et argumenterons que notre approche conduit à une synthèse élégante et canonique.

On verra finalement que la σ-équivalence discutée plus haut nous suggère un peu plus de partage, nous faisant faire une étape de plus vers la pleine paresse [3].

3.1. Réduction linéaire de tête faible

La définition qu'on a donnée de la réduction linéaire de tête au paragraphe 2.4 est une réduction dite *forte*, car elle réduit sous les λ. Ceci n'est pas forcément désirable, pour plusieurs raisons. D'abord, les implémentations « réalistes » du λ-calcul sont toutes en réduction faible, en particulier la KAM. Si l'on veut se baser par la suite sur cette implémentation, mieux vaut nous restreindre de même à une réduction faible. Ensuite, la réduction forte est l'antithèse de l'appel par nécessité : quel est l'intérêt de calculer sous un λ si l'on en a pas encore besoin ? Dans cette optique, mieux vaut présenter un calcul faible.

On peut contraindre la réduction λ_{lh} à ne pas réduire sous les λ en restreignant les contextes d'évaluation sous lequels elle peut agir ; on ne pourrait descendre que sous les contextes de la réduction de tête faible :

$$E^w ::= [\cdot] \mid E^w\, u$$

Cependant, cette contrainte est par trop stricte pour que la réduction puisse continuer à fonctionner correctement. En effet, considérons la règle :

$$E_1^w[\mathcal{C}[\lambda x.\, E_2^w[x]]\, u] \to E_1^w[\mathcal{C}[\lambda x.\, E_2^w[u]]\, u] \qquad (\textsc{Presque})$$

et visualisons-la au travers par exemple de la KAM, qui implémente bien une réduction faible. La décomposition du contexte ne doit se faire que *via* les règles Push et Pop, ce qui est compatible avec la restriction aux E^w. Cependant, il existe des configurations créant des contextes de clôture qui ne relèvent pas de ce motif. Que penser de la situation suivante ?

$$t \quad \equiv \quad \mathcal{C}[\lambda x.\, \lambda z_1.\, \ldots z_n.\, E^w[x]]\, u\, r_1\, \ldots r_n$$

Dans ce cas, les couples de radicaux premiers $\{z_i \leftarrow r_i\}$ sont tous transparents dans la réduction de la KAM et ne participent qu'à la création de la clôture de $E^w[x]$. On a en effet :

$$\langle (t, \sigma) \mid \pi \rangle \longrightarrow^*_{\textsc{Push},\textsc{Pop}} \langle (E^w[x], \sigma + [\mathcal{C}]_\sigma + (x := (u, \sigma)) + (\vec{z} := (\vec{r}, \sigma))) \mid \pi \rangle$$

Malgré tout, la règle Presque ne capture pas cette situation, parce que E_2^w interdit la présence de λ-abstractions. En d'autres termes, il faut autoriser la présence de lieurs dans les contextes considérés *sous réserve* qu'ils aient été compensés par suffisamment d'applications antérieures, c'est-à-dire qu'ils participent d'un radical premier.

Les contextes de clôture ne sont pas suffisants pour capturer ce genre de réduction, car il faut pouvoir décomposer le contexte environnant en deux parties. Sur cette base, on pourrait vouloir utiliser des paires de contextes applicatifs et lieurs de la forme suivante :

$$\begin{array}{rcccl} E^@ & ::= & [\cdot] & \mid & E^@\, u \quad (\equiv E^w) \\ E^\lambda & ::= & [\cdot] & \mid & \lambda x.\, E^\lambda \end{array}$$

Les contextes applicatifs définissent un trop-plein d'applications, et les lieurs consomment le surplus. On ne considérerait alors que les paires de contextes qui, une fois composés, arriveraient à l'équilibre.

Définition 8 (Comblement)**.** Soient $E^@$ et E^λ des contextes respectivement applicatif et lieur. On dit que $E^@$ comble E^λ (noté $E^@ \ni E^\lambda$) si $E^@$ contient plus d'applications que E^λ ne contient de lieurs, ou, formellement :

$$\frac{}{E^@ \ni [\cdot]} \qquad\qquad \frac{E^@ \ni E^\lambda}{E^@\, u \ni \lambda x.\, E^\lambda}$$

La règle modifiée deviendrait alors :

$$E^@[\mathcal{C}[\lambda x.\, E^\lambda[E^w[x]]]\, u] \to E^@[\mathcal{C}[\lambda x.\, E^\lambda[E^w[u]]]\, u] \qquad \text{si } E^@ \ni E^\lambda.$$

Cela ne suffit pas non plus, car il faut malgré tout être transparent pour les contextes de clôtures. Cependant, cela peut être facilement réparé. Nous introduisons pour cela les contextes applicatifs et lieurs *à clôture près* définis comme suit :

$$\begin{array}{rcccl} \mathcal{C}^@ & ::= & \mathcal{C} & \mid & \mathcal{C}[\mathcal{C}^@\, u] \\ \mathcal{C}^\lambda & ::= & \mathcal{C} & \mid & \mathcal{C}[\lambda x.\, \mathcal{C}^\lambda] \end{array}$$

D'une certaine façon, ces derniers contextes sont une manière de rendre insensibles les contextes E^λ et $E^@$ aux contextes de clôture en y introduisant de fines tranches de clôtures. On pourrait aussi l'expliquer sous la forme de systèmes de transition, comme la clôture, à clôtures près [3], des contextes évoqués précédemment : chaque étape Push (respectivement Pop) de la KAM a désormais le droit de faire autant de réductions qu'elle veut sous réserve que l'effet global se résume à faire grossir la clôture du terme courant.

3. Sans mauvais jeux de mots.

La notion de comblement se relève trivialement à clôture près, et l'on peut finalement définir la réduction linéaire de tête faible. Par facilité de notation, on écrira $\mathcal{C}^w$ pour un contexte applicatif à clôture près que l'on ne désire pas composer avec un contexte lieur [4].

Définition 9 (Réduction λ_{wlh})**.** On définit le calcul de réduction linéaire de tête faible λ_{wlh} par la règle :

$$\mathcal{C}^{@}[(\mathcal{C}[\lambda x.\, \mathcal{C}^{\lambda}[\mathcal{C}^{w}[x]]])\, u] \to_{\lambda_{wlh}} \mathcal{C}^{@}[(\mathcal{C}[\lambda x.\, \mathcal{C}^{\lambda}[\mathcal{C}^{w}[u]]])\, u] \qquad \text{si } \mathcal{C}^{@} \ni \mathcal{C}^{\lambda}$$

Remarque 4. Si $\mathcal{C}^{@} \ni \mathcal{C}^{\lambda}$, alors $\mathcal{C}^{@}[\mathcal{C}^{\lambda}]$ est un contexte applicatif, à clôture près.

On peut donner une définition alternative basée sur la version historique de Danos-Regnier, qui hérite des mêmes artéfacts de présentation que la version forte.

Définition 10 (Réduction wlh, version historique)**.** On dit que t se réduit sur r par réduction linéaire de tête faible, noté $t \to_{wlh} r$, si les deux conditions suivantes sont réunies :

1. t se réduit par réduction linéaire de tête sur r ;
2. la liste des lambdas de tête de t est vide.

On voit que cette présentation traite implicitement des contextes lieurs : ne pas avoir de lambdas de tête correspond précisément à avoir ses sous-contextes lieurs comblés. Les deux notions de réduction faible sont alors équivalentes.

Proposition 8. *Le calcul λ_{wlh} et la réduction linéaire de tête faible historique coïncident :*

$$t \to_{\lambda_{wlh}} r \qquad \textit{ssi} \qquad t \to_{wlh} r$$

3.2. Passage en appel par « valeur »

On définit maintenant une variante en appel par valeur de la réduction linéaire de tête faible, dans un but de diminution des calculs. On ne veut en effet pas devoir dupliquer des calculs déjà faits lors du calcul d'un argument.

Pour ce faire, on va restreindre les contextes provoquant la substitution à ne réagir qu'en présence de valeurs. En outre, dans le même esprit d'insensibilité aux clôtures qui nous a guidés jusqu'ici, nous allons aussi considérer des valeurs à clôture près.

Les valeurs v sont les valeurs habituelles du λ-calcul, c'est-à-dire :

$$v ::= \lambda x.\, t.$$

Les valeurs à clôture près w sont définies directement comme :

$$w ::= \mathcal{C}[v].$$

Pour passer de l'appel par nom à l'appel par valeur, il faut rajouter les contextes forçant les valeurs ; de la même manière, on considère maintenant les contextes qui forcent les valeurs à clôture près . La construction est systématique en appliquant l'encodage habituel de l'appel par valeur.

$$E^{v} ::= [\cdot] \mid E^{v}\, u \mid \mathcal{C}^{@}[(\mathcal{C}[\lambda x.\, \mathcal{C}^{\lambda}[E_1^{v}[x]]])\, E_2^{v}] \mid \mathcal{C}[E^{v}] \qquad \text{où } \mathcal{C}^{@} \ni \mathcal{C}^{\lambda}.$$

Notons que, par souci de lisibilité, on a explicité la possibilité de raisonner à contexte près par une règle dédiée $\mathcal{C}[E^{v}]$, mais on pourrait tout aussi bien procéder au sandwichage en l'*inlinant* dans la définition, comme c'est le cas dans la section précédente.

La réduction linéaire de tête en appel par valeur s'obtient alors directement :

4. Les contextes applicatifs et faibles étant strictement identiques, la notation fait sens.

Définition 11. L'appel par valeur linéaire de tête est défini par la réduction :

$$\mathcal{C}^{@}[(\mathcal{C}[\lambda x.\, \mathcal{C}^{\lambda}[E^{v}[x]]])\, w] \to_{\lambda_{wlv}} \mathcal{C}^{@}[(\mathcal{C}[\lambda x.\, \mathcal{C}^{\lambda}[E^{v}[w]]])\, w] \qquad \text{si } \mathcal{C}^{@} \ni \mathcal{C}^{\lambda}$$

On peut remarquer que la règle de réduction n'a pas été beaucoup modifiée ; l'essentiel de la difficulté se trouve dans le choix idoine des contextes.

Ce calcul, quoique désigné sous le nom d'appel par valeur, impémente déjà un appel par nécessité : la réduction a beau ne substituer que des valeurs, elle ne se lance dans le calcul d'un argument que si d'aventure elle a rencontré une variable lui correspondant en position de hoc (adapté à l'appel par valeur). On donne la réduction sur notre exemple au long cours :

$$\begin{array}{lll}
\Delta\,(I\,I) & \equiv & (\lambda x.\, x\, x)\,((\lambda y.\, y)\,(\lambda z.\, z)) \\
& \to_{wlv} & (\lambda x.\, x\, x)\,((\lambda y.\, \lambda z_0.\, z_0)\,(\lambda z.\, z)) \\
& \to_{wlv} & (\lambda x.\, (\lambda y_1.\, \lambda z_1.\, z_1)\,(\lambda z_2.\, z_2)\, x)\,((\lambda y.\, \lambda z_0.\, z_0)\,(\lambda z.\, z)) \\
& \to_{wlv} & (\lambda x.\, (\lambda y_1.\, \lambda z_1.\, x)\,(\lambda z_2.\, z_2)\, x)\,((\lambda y.\, \lambda z_0.\, z_0)\,(\lambda z.\, z)) \\
& \to_{wlv} & (\lambda x.\, (\lambda y_1.\, \lambda z_1.\, ((\lambda y_2.\, \lambda z_3.\, z_3)\,(\lambda z_4.\, z_4)))\,(\lambda z_2.\, z_2)\, x)\,((\lambda y.\, \lambda z_0.\, z_0)\,(\lambda z.\, z)) \\
& \to_{gc} & I
\end{array}$$

On peut bien constater dans la première transition le fait que la réduction va s'opérer dans l'argument qui a été demandé par x, avant de renvoyer une valeur qui sera substituée par la deuxième réduction.

3.3. Partage de clôtures

Le fait de n'autoriser à substituer que des valeurs est un premier pas pour implémenter l'appel par nécessité, mais il n'est pas suffisant pour atteindre les calculs considérés dans la littérature. En effet, si l'on observe bien la règle de réduction de la section précédente, on peut être témoin d'une duplication inutile des calculs :

$$(\mathcal{C}'[\lambda x.\, E^{v}[x]])\,\mathcal{C}[v] \to_{\lambda_{wlv}} (\mathcal{C}'[\lambda x.\, E^{v}[\mathcal{C}[v]]])\,\mathcal{C}[v]$$

Ici, le contexte de clôture $\mathcal{C}$ a été copié, ce qui provoquera le recalcul de ses termes liés si jamais ils sont utilisés au cours de la réduction. Ce phénomène n'est pas visible sur notre exemple $\Delta\,(I\,I)$, car le premier I ne duplique pas son argument. Au contraire, le terme suivant produirait une duplication de contexte inutile :

$$(\lambda x.\, (x\, I)\, x)\,((\lambda y.\, \lambda z.\, y)\,(I\,I))$$

car le terme à droite de l'application est déjà une valeur à clôture près, et il utilise un argument de sa clôture, le terme $(I)\,I$. Lors de la substitution, ce calcul est copié tel quel, ce qui fait que lors du calcul du corps de l'abstraction, chaque appel à x nécessitera de le recalculer. Le *call-by-need* d'Ariola et Felleisen évite cet écueil grâce à la règle Assoc.

On peut résoudre ce problème d'une façon simple et élégante, d'une manière similaire à ce que fait la règle Assoc : au cours de la substitution d'une valeur, on procède aussi à une extrusion de la clôture de cette valeur. Il n'y a pas besoin de procéder à un raffinement des contextes car tout y est déjà présent.

Le calcul résultant est précisé ci-dessous.

Définition 12. L'appel par valeur linéaire de tête avec partage est défini par l'unique réduction :

$$\mathcal{C}^{@}[(\mathcal{C}[\lambda x.\, \mathcal{C}^{\lambda}[E^{v}[x]]])\, \mathcal{C}'[v]] \to_{\lambda_{wlv}} \mathcal{C}^{@}[\mathcal{C}'[(\mathcal{C}[\lambda x.\, \mathcal{C}^{\lambda}[E^{v}[v]]])\, v]] \qquad \text{si } \mathcal{C}^{@} \ni \mathcal{C}^{\lambda}$$

avec les conventions habituelles pour éviter la capture de variables de $\mathcal{C}'$.

Valeurs	v	::=	$\lambda x.\, t$
Réponses	a	::=	$A[v]$
Contextes de réponses	A	::=	$[\cdot] \mid A_1[\lambda x.\, A_2]\, u$
Contextes de réponses internes	A^λ	::=	$[\cdot] \mid A[\lambda x.\, A^\lambda]$
Contextes de réponses externes	$A^@$	::=	$[\cdot] \mid A[A^@]\, u$
Contextes	E	::=	$[\cdot] \mid E\, u \mid A[E]$
			$\mid A^@[A[\lambda x.\, A^\lambda[E[x]]]\, E]$ avec $A^@[A^\lambda] \in A$

$$A^@[A_1[\lambda x.\, A^\lambda[E[x]]]\, A_2[v]] \longrightarrow A^@[A_1[A_2[(A^\lambda[E[x]])\{x := v\}]]] \quad \text{si } A^@[A^\lambda] \in A \qquad (\beta_{\text{cf}})$$

FIGURE 2 – *Call-by-need* de Chang-Felleisen

3.4. Comparaison avec les calculs en appel par nécessité

Assez remarquablement, le calcul ainsi obtenu est précisément le *call-by-need* de Felleisen et Chang, à l'exception qu'il utilise une substitution linéaire au lieu de la substitution destructive habituelle du λ-calcul. On rappelle ce calcul à la figure 3.4.

On se rend bien compte en particulier que les contextes de réponse A correspondent à nos contextes de clôture, et que les variantes internes et externes correspondent aux contextes lieurs et applicatifs respectivement.

De petites différences apparaissent, en particulier au niveau du comblement. Cette condition du calcul de Chang-Felleisen est exprimée comme une égalité, mais les deux formalismes sont équivalents, puisqu'il s'agit essentiellement de réarranger la manière dont on a découpé les contextes.

On peut exprimer β_{cf} dans notre formalisme de manière quasiment transparente.

Définition 13 (Chang-Felleisen *revisited*)**.** Le λ-calcul en appel par nécessité de Felleisen et Chang est décrit par la règle de réduction :

$$\mathcal{C}^@[(\mathcal{C}[\lambda x.\, \mathcal{C}^\lambda[E^v[x]]])\, \mathcal{C}'[v]] \to_{\beta_{\text{cfr}}} \mathcal{C}^@[\mathcal{C}[\mathcal{C}'[(\mathcal{C}^\lambda[E^v[v]])\{x := v\}]]] \qquad \text{si } \mathcal{C}^@ \ni \mathcal{C}^\lambda.$$

Remarque 5. On notera que le calcul de Chang et Felleisen emboîte les clôtures dans l'ordre opposé de notre calcul. Alors que l'utilisation d'une substitution non linéaire et destructrice permet ce choix, l'utilisation d'une substitution linéaire nous force à faire le choix présenté plus haut.

Morale 1. La réduction linéaire de tête faible par valeur avec partage de clôtures *est* un λ-calcul en appel par nécessité.

3.5. Vers la pleine paresse

Évaluation pleinement paresseuse. Les calculs considérés jusque là ne capturent pas ce que l'on appelle la pleine paresse. Par exemple, pour reprendre l'exemple initial, l'évaluation de $(I)\, I$ dans $(\Delta)(I)I$ est partagée dans les calculs considérés jusque là, mais elle n'est pas partagée dans $(\Delta)\lambda x.\, ((I)\, I)\, x$ (obtenu par η-expansion du premier terme).

L'interpréteur de Wadsworth va quant à lui partager le calcul de $(I)\, I$; ce niveau de partage des calculs est désigné sous l'appellation d'évaluation pleinement paresseuse (ou *fully lazy*).

Ariola et Felleisen [3] proposent un calcul pour la pleine paresse dans la dernière section de leur article sous le nom de calcul de Wadsworth :

Définition 14 (Sous-expressions libres)**.** Soit t une abstraction et u un sous-terme de t. On dira que u est une expression libre de t si aucune variable libre de u n'est liée dans t et si u est une application $(v)\, w$.

On dira qu'un expression libre de t, u, est une expression libre maximale si aucune expression libre de t ne contient strictement u.

Étant donnés des termes t et $\vec{u} \equiv u_1, \dots u_n$, $\mathrm{mfe}(t, \vec{u})$ signifiera que les u_i sont des expressions libres maximales de t. La métavariable V_{mfe} dénotera les valeurs ne contenant aucune expression libre (maximale).

Définition 15 (Calcul de Wadsworth)**.** Le calcul de Wadsworth est obtenu en modifiant le calcul d'Ariola et Felleisen (présenté en définition 1) par la décomposition de la règle DEREF en deux règles $\text{DEREF}_{\text{wad}}$ et MFE_{wad} :

$$\begin{array}{lllll}
(\text{DEREF}_{\text{wad}}) & (\lambda x.\, E[x])\, V_{\mathrm{mfe}} & \to & (\lambda x.\, E[V_{\mathrm{mfe}}])\, V_{\mathrm{mfe}} \\
(\text{MFE}_{\text{wad}}) & (\lambda x.\, E[x])\, \lambda y.\, C[\vec{t}] & \to & (\lambda x.\, E[x])\, (\lambda \vec{z} y.\, C[\vec{z}])\, \vec{t} \\
 & & & \text{si } \mathrm{mfe}(\lambda y.\, C[\vec{t}], \vec{t}) \text{ et } \nexists u, \mathrm{mfe}(\lambda y.\, C[\vec{z}], u) \\
(\text{LIFT}) & (\lambda x.\, A)\, t\, u & \to & (\lambda x.\, (A)\, u)\, t \\
(\text{ASSOC}) & (\lambda x.\, E[x])\, (\lambda y.\, A)\, t & \to & (\lambda y.\, (\lambda x.\, E[x])\, A)\, t
\end{array}$$

Le calcul proposé par Chang et Felleisen, de même que le calcul synthétisé dans les précédentes sections, ne capture pas la pleine paresse. Même s'ils effectuent bien du partage de clôtures, ces calculs ne vont pas partager les calculs qui pourraient être contenus dans une valeur.

Vers une évaluation pleinement paresseuse. Pourtant, la σ-équivalence nous apporte un éclairage intéressant sur les expressions libres. La règle suivante (avec les conditions usuelles sur les variables liées) :

$$\lambda x.\, (\lambda y.\, t)\, u \to (\lambda y.\, \lambda x.\, t)\, u$$

est une instance de la σ-équivalence de Regnier. Le contenu calculatoire de cette règle consiste à prendre une valeur contenant une clôture, à exposer cette clôture en la faisant commuter avec l'abstraction et ainsi à la rendre accessible au partage de clôtures.

Puisqu'on a l'hypothèse que u ne contient pas x comme variable libre, on voit que (dans le cas où u est une application), u est une sous-expression libre maximale du terme considéré et qu'on est tout près de la réduction mfe puisque celle-ci nous donnerait :

$$\lambda x.\, (\lambda y.\, t)\, u \to_{\text{MFE}_{\text{wad}}} (\lambda z.\, \lambda x.\, (\lambda y.\, t)\, z)\, u$$

Il est donc tentant d'incorporer de la détection de sous-expression libre à base de σ-équivalence dans notre calcul provenant de la réduction linéaire de tête. Nous n'irons pas plus loin dans cette direction dans le présent article car le cadre que nous avons adopté jusque là, celui d'un calcul avec une unique réduction macroscopique se prête difficilement à mener cette idée jusqu'au bout.

En revanche, la dérivation d'un calcul en appel par nécessité depuis une version atomique de la réduction linéaire de tête devrait nous permettre d'être (presque) pleinement paresseux.

4. Conclusion

Dans cet article, nous avons dérivé de manière systématique un calcul en appel par nécessité à partir de la réduction linéaire de tête en trois étapes : restriction à une réduction faible, passage de valeurs, partage de clôtures. Au passage, nous avons reformulé la réduction linéaire de tête qui, dans la littérature, est présentée d'une manière relativement peu praticable.

Les ingrédients pour aboutir à ce résultat ont été :
– la clarification du fonctionnement de la réduction linéaire de tête,

- la mise en évidence de la notion de contexte de clôture et le fait de placer cette structure au centre de notre construction : toutes les constructions que nous donnons sont "closes" pour les contextes de clôtures,
- la prise en sérieux de la σ-équivalence.

Réduction macroscopique *vs.* **réduction microscopique.** Partant d'un calcul avec une réduction macroscopique, nous arrivons à un appel par nécessité macroscopique (c'est-à-dire définie par un unique axiome). Nous aurions pu prendre un autre chemin, que nous explorerons à l'avenir : on peut sans aucun problème intégrer les règles de la σ-équivalence dans le calcul de la réduction linéaire de tête de manière à définir une réduction linéaire de tête progressive. Dans ce cas, nous conjecturons qu'on serait arrivé précisément sur le calcul d'Ariola et Felleisen. L'approche microscopique semble également prometteuse pour s'approcher de la pleine paresse.

Liens entre réduction linéaire de tête et appel par nécessité. Les liens entre réduction linéaire de tête et appel par nécessité sont frappants. Il est tout aussi frappant de voir qu'on ne trouve pas de lien dans la littérature entre appel par nécessité et réduction linéaire de tête. Par exemple dans l'introduction d'un article récent de Danvy et al. [10], un calcul en appel par nom avec let est présenté avec un objectif d'illustration. Ce calcul est essentiellement un calcul pour la réduction linéaire de tête.

Réseaux de preuve et call-by-need. À notre sens, cette connexion est potentiellement fructueuse : l'expression de l'appel par nécessité à partir de la réduction linéaire de tête va nous permettre de mener une analyse de l'appel par nécessité dans les réseaux de preuve, en allant possiblement jusqu'à l'analyse de la pleine paresse comme le suggère la section précédente. Les liens entre logique linéaire et appel par nécessité sont à creuser car ils vont bien au-delà de l'ébauche qu'on trouve dans [14].

Dans ce sens, il sera également utile de considérer les liens entre notre approche et le λ-calcul structurel d'Accattoli et Kesner [1].

Vers le contrôle. Une autre ligne de travail à venir consiste à considérer l'extension au contrôle. Dans des travaux précédents, le second auteur a déjà proposé une extension de l'appel par nécessité au contrôle [5, 2]. La méthodologie était différente même si, là encore, la méthode reposait sur la théorie de la preuve. Il s'agissait de partir de calculs construits sur le calcul des séquents pour les adapter à l'appel par nécessité. Dans cette conception, l'intégration du contrôle est transparente mais c'est la spécification de ce qu'est la paresse qui est loin d'être évidente et pas forcément canonique. (En particulier on observait dans ces travaux que différentes manières d'intégrer l'appel par nécessité et le contrôle résultaient en des calculs différents.) Ici, on est dans une configuration différente : la spécification de l'appel par nécessité est maintenant solidement justifié et il va falloir intégrer le contrôle.

Références

[1] B. Accattoli and D. Kesner. The structural *lambda*-calculus. In *Computer Science Logic, CSL*, volume 6247 of *Lecture Notes in Computer Science*, pages 381–395. Springer, 2010.

[2] Z. M. Ariola, P. Downen, H. Herbelin, K. Nakata, and A. Saurin. Classical call-by-need sequent calculi : The unity of semantic artifacts. In *Functional and Logic Programming Symposium, FLOPS 2012*, volume 7294 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2012.

[3] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3) :265–301, 1997.

[4] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In R. K. Cytron and P. Lee, editors, *Symposium on Principles of Programming Languages, POPL*, pages 233–246. ACM Press, 1995.

[5] Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In *Typed Lambda Calculi and Applications*, volume 6690 of *lncs*, 2011.

[6] S. Chang and M. Felleisen. The call-by-need lambda calculus, revisited. *European Symposium on Programming, ESOP*, abs/1201.3907, 2012.

[7] P.-L. Curien and H. Herbelin. The duality of computation. In *International Conference on Functional Programming, ICFP*, 2000.

[8] V. Danos, H. Herbelin, and L. Regnier. Game semantics & abstract machines. In *Logic in Computer Science, LICS*, pages 394–405, 1996.

[9] V. Danos and L. Regnier. Head linear reduction. Technical report, 2004.

[10] O. Danvy, K. Millikin, J. Munk, and I. Zerny. Defunctionalized interpreters for call-by-need evaluation. In *Functional and Logic Programming Symposium, FLOPS2010*, 2010.

[11] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50 :1–102, 1987.

[12] M. Hyland and L. Ong. On full abstraction for PCF. *Information and Computation*, 163(2) :285–408, Dec. 2000.

[13] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3) :199–207, 2007.

[14] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Electronic Notes in Theoretical Computer Science*, 1 :370–392, 1995.

[15] J. Maraist, M. Odersky, and P. Wadler. The call-by-need λ-calculus. *Journal of Functional Programming*, 8(3) :275–317, 1998.

[16] L. Regnier. *Lambda-calcul et réseaux*. PhD thesis, Univ. Paris VII, 1992.

[17] L. Regnier. Une équivalence sur les lambda-termes. *Theoretical Computer Science*, 126 :281–292, 1994.

[18] C. P. Wadsworth. *Semantics and pragmatics of the lambda-calculus*. PhD thesis, Programming Research Group, Oxford University, 1971.

Communications courtes

Traduction prouvée de HOL4 vers le premier ordre

Thibaut Gauthier, Chantal Keller et Michael Norrish

Résumé

Nous présentons une traduction de la logique d'ordre supérieur polymorphe vers la logique du premier ordre monomorphe implantée et prouvée correcte dans l'assistant de preuves HOL4, faisant notamment intervenir de nouvelles techniques de monomorphisation. Elle permet d'interfacer HOL4 avec des prouveurs du premier ordre en vue de décharger certains buts à ces prouveurs automatiques, tout en ayant confiance dans la traduction. Des expériences utilisant le prouveur du premier ordre avec arithmétique Beagle montrent son expressivité (combinaison de raisonnements propositionnel et arithmétique) et l'efficacité de notre monomorphisation.

Le petit guide du bouturage

Jean-Christophe Filliatre

Résumé

Un langage de programmation fonctionnel s'avère un outil de choix pour manipuler des arbres de façon purement applicative. D'une part, les arbres y sont représentés de manière naturelle par un type algébrique et manipulés de façon concise et élégante par le filtrage. D'autre part, le typage statique assure la bonne formation des arbres et exclut tout risque d'avoir omis un cas de figure dans une définition de fonction.

Si on cherche en revanche à manipuler des arbres mutables, c'est-à-dire qui peuvent être modifiés en place, tout en restant dans le cadre d'un langage fonctionnel, il semble qu'on doive renoncer partiellement à ces atouts. C'est d'autant plus regrettable que l'immense majorité de la littérature algorithmique ne considère que des arbres mutables. Dans cet article, nous explorons différentes représentations d'arbres mutables dans le contexte du langage OCaml. Nous les comparons en termes de performance, mais aussi en termes d'élégance et de sûreté.

Parsifal : une solution pour écrire rapidement des *parsers* binaires robustes et efficaces

Olivier Levillain[1,2]

1: Agence nationale de la sécurité des systèmes d'information (ANSSI)
2: Télécom Sud Paris

Dans le cadre de ses activités d'expertise, le laboratoire sécurité des réseaux et protocoles de l'ANSSI est amené à étudier divers protocoles de communication. L'étude fine de ces protocoles passe par l'utilisation de *parsers* (ou dissecteurs) permettant d'analyser les messages binaires échangés lors d'une exécution du protocole. L'expérience a montré qu'il fallait disposer d'outils robustes et maîtrisés pour étudier et comprendre les comportements d'un protocole donné, en particulier pour en détecter et caractériser les anomalies. En effet, les implémentations disponibles sont parfois limitées (refus de certaines options), laxistes (acceptation silencieuse de paramètres erronés) ou fragiles (terminaison brutale du programme pour des valeurs inattendues, qu'elles soient licites ou non). Ce constat nous a conduit au développement d'outils, l'objectif étant de développer *rapidement* des dissecteurs *robustes* et *performants*. Ce document décrit brièvement Parsifal, une implémentation générique de *parsers* binaires reposant sur le pré-processeur `camlp4` d'OCaml.

Afin de mieux comprendre un protocole et la manière dont il est utilisé *in vivo*, le laboratoire s'intéresse notamment à l'analyse de grands volumes de données issus de mesures réalisées sur internet. Le point de départ de nos travaux sur les *parsers* binaires est un ensemble important de traces réseau contenant des échanges suivant le protocole TLS (*Transport Layer Security* [3]) mises à disposition par l'EFF (*Electronic Frontier Foundation*) [1]. Ces mesures ont fait l'objet d'une publication [2]. L'analyse de ces données pose plusieurs difficultés. Tout d'abord, les fichiers à analyser représentent un volume conséquent (180 Go dans le cas de notre analyse de TLS). Ensuite, les informations à extraire sont contenues dans des messages de structures complexes. Enfin, il s'agit de données brutes, non filtrées, dont la qualité, voire l'innocuité, laisse parfois à désirer.

Description de Parsifal

Parsifal est issu des besoins identifiés et de l'expérience acquise dans l'écriture de *parsers* pour des formats binaires. Il s'agit d'une implémentation générique de *parsers* reposant sur un pré-processeur `camlp4` et sur une bibliothèque auxiliaire.

Le concept de base de Parsifal est la définition de « types enrichis », les $\mathcal{PT}$ypes, qui sont simplement des types OCaml quelconques pour lesquels certaines fonctions sont fournies. Ainsi, un $\mathcal{PT}$ype est défini par un type OCaml `t` décrivant le contenu à *parser*, par une fonction pour disséquer les objets depuis une chaîne de caractères (`parse_t`) et par des fonctions pour exporter les objets sous forme binaire (`dump_t`) ou dans une représentation haut niveau utile aux fonctions d'affichage (`value_of_t`).

On peut distinguer trois sortes de $\mathcal{PT}$ypes. Tout d'abord, la bibliothèque standard fournit des $\mathcal{PT}$ypes de base (entiers, chaînes de caractère, listes, etc.). Ensuite, il estpossible de construire des $\mathcal{PT}$ypes à partir de mots clés tels que `struct`, `union`, `enum` ; pour ceux-ci, une description suffit au pré-processeur pour générer automatiquement le type OCaml et les fonctions correspondantes. Enfin, dans certains cas, il est nécessaire d'écrire le type OCaml et les fonctions `parse_t`, `dump_t` et `value_of_t`

à la main, pour gérer des cas particuliers. Pour illustrer les deux premiers types de $\mathcal{PT}$ypes, voici une implémentation rudimentaire des messages TLS à l'aide de Parsifal :

```
enum tls_content_type (8, Exception) =
  | 0x14 -> CT_ChangeCipherSpec      | 0x15 -> CT_Alert
  | 0x16 -> CT_Handshake             | 0x17 -> CT_ApplicationData

union record_content (Unparsed_Record) =
  | CT_Alert             -> Alert of array(2) of uint8
  | CT_Handshake         -> Handshake of binstring
  | CT_ChangeCipherSpec -> ChangeCipherSpec of uint8
  | CT_ApplicationData  -> ApplicationData of binstring

struct tls_record = {
  content_type : tls_content_type;
  record_version : tls_version;
  record_content : container[uint16] of record_content (content_type)
}
```

Le dernier bloc de code décrit ce qu'est un *record* TLS, un enregistrement (décrit à l'aide du mot clé `struct`) contenant quatre champs : le type du contenu, la version du protocole, la taille du contenu et le contenu lui-même. Pour le premier champ, il existe 4 types de contenu, qui sont décrits par l'énumération `tls_content_type` (anoncée par le mot clé `enum` du premier bloc). Cette valeur est stockée sur un entier 8 bits, et si la lecture de ce champ donne une valeur non énumérée, une exception sera levée ; c'est le sens des paramètres de l'énumération (`8` et `Exception`).

La version TLS est stockée sur 16 bits : on utilise donc le $\mathcal{PT}$ype prédéfini `uint16`. Comme il existe certaines versions connues, on pourrait utiliser une énumération ici également, avec un comportement plus laxiste face aux valeurs inconnues (ajout d'un constructeur avec `UnknownVal`) :

```
enum tls_version (16, UnknownVal UnknownVersion) =
  | 0x0002 -> SSLv2       | 0x0300 -> SSLv3
  | 0x0301 -> TLSv1       | 0x0302 -> TLSv1_1
  | 0x0303 -> TLSv1_2
```

Les deux derniers champs du *tls_record* sont décrits ensemble par un conteneur dont la longueur, variable, tient sur 16 bits. Le contenu du message lui-même est décrit par le $\mathcal{PT}$ype `record_content`, qui prend un argument (`content_type`). En effet, `record_content` est une `union`, dont le contenu dépend d'un discriminant, ici le type de contenu. Par exemple, une alerte contient 2 octets.

Bilan de deux ans d'écriture de *parsers* binaires

Après avoir écrit plusieurs implémentations dans différents langages (Python, C++, OCaml), nous avons développé Parsifal, une implémentation générique de *parsers* binaires reposant sur un pré-processeur, qui remplit nos besoins : possibilité d'exprimer des formats complexes, rapidité d'écriture, robustesse et performances. Plusieurs protocoles réseau ont déjà été (au moins partiellement) décrits à l'aide de Parsifal (TLS, DNS, BGP...), ainsi que plusieurs formats de fichiers (TAR, PNG, JPG...).

Cette courte présentation de Parsifal n'a pas fait état d'autres constructions pratiques pour le développeur : gestion des structures ASN.1 DER, conteneurs personnalisés, champs de bits... Le projet est disponible en tant que logiciel libre sur `https://github.com/ANSSI-FR/parsifal`.

Références

[1] P. Eckersley and J. Burns. An Observatory for the SSLiverse, Talk at Defcon 18, 2010.

[2] O. Levillain, A. Ébalard, H. Debar, and B. Morin. One Year of SSL Measurement. ACSAC, 2012.

[3] T. Dierks and E. Rescorla. The Transport Layer Security Protocol Version 1.2. RFC 5246, 2008.