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

Julien Signoles[1]

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

**Résumé**

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

## 1. Introduction

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

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

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

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

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

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

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

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

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

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

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

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

[1] La version distribuée de *Frama-C* comprend plus de cent mille lignes de code *OCaml*.

[2] En réalité, le problème étant indécidable, il ne supprime qu'une sous-approximation des parties non pertinentes du programme afin d'être correct.

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

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

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

# 3. Projets et états

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

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

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

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

## 3.1. Mécanisme général

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

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

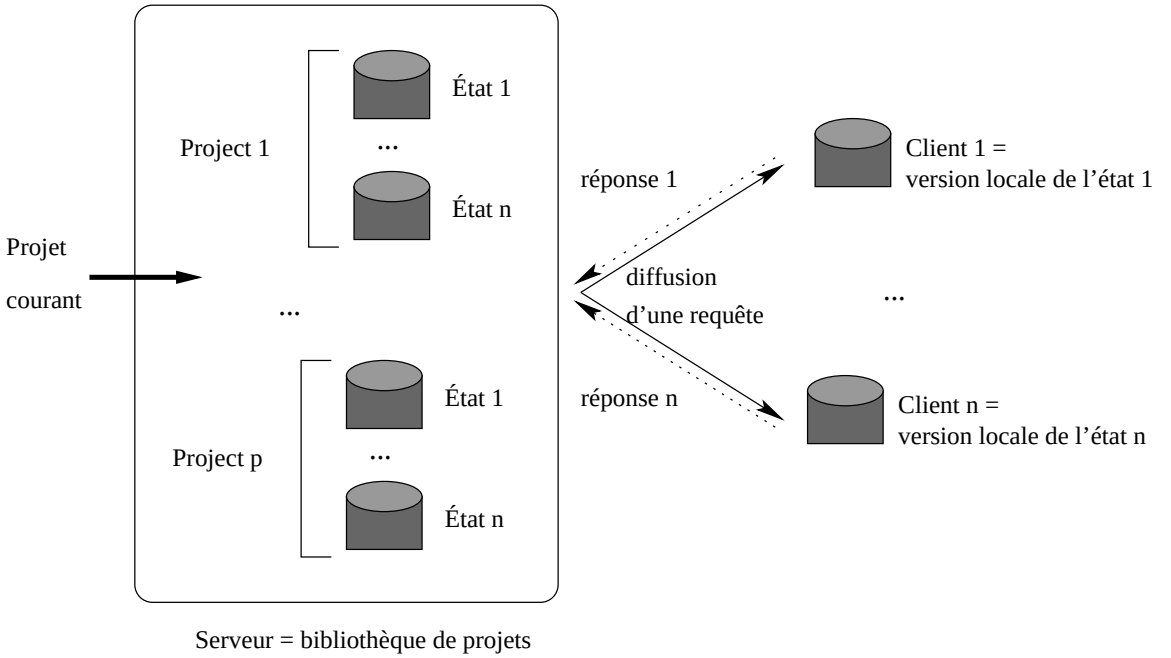


FIG. 1 – Fonctionnement de type client-serveur avec *broadcast*.

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

## 3.2. Enregistrement d'un nouvel état

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 3.3. Opérations de base sur les projets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### 3.4. Cloisonnement des projets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# 4. Modules de plus haut-niveau

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

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

module type REF_OUTPUT = sig
```

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

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

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

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

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

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

## 5. Sélections et dépendances d'états

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

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

Afin de pouvoir créer de tels ensembles, il est nécessaire de pouvoir manipuler les états et, en particulier, de pouvoir les comparer. Dans ce but, le type `state_operations`, introduit page 5 et représentant le type des états, contient en réalité un nom (une chaîne de caractères), fourni au moment de l'application du foncteur `Register`, qui permet d'identifier les états de manière unique[8].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# 6. Sérialisation et hashconsing

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

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

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

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

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

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

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

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

# 7. Types de données

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# 8. Conclusion

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

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

# 9. Remerciements

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

# Références

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# A. Preuve de la propriété de cloisonnement des projets

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

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

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

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

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

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

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

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

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

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

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

– **Cas de la fonction** `create`

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

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

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

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

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

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

– **Cas de la fonction** `copy`

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

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