Transformations certifiées de programmes impératifs

Sandrine Blazy

décembre 2002 – rapport CEDRIC numéro 398– 02

Transformations certifiées de programmes impératifs

Sandrine Blazy

CEDRIC IIE,
18 allée Jean Rostand
91 025 Évry cedex
blazy@iie.cnam.fr

Résumé

Nous présentons une transformation de programmes impératifs inspirée de l'évaluation partielle et ayant été utilisée pour faciliter la compréhension d'applications scientifiques. Ce travail a été complètement formalisé en Coq. Partant d'une spécification du langage que nous traitons, nous avons spécifié de façon abstraite notre analyse de code et avons prouvé sa correction sémantique. Afin de modéliser les états-mémoire de façon abstraite, nous avons défini une couche de bas niveau modélisant la notion générique de table d'association ainsi que des opérations sur ces tables et contenant également des preuves de propriétés relatives aux opérations. Nous montrons que ces tables génériques sont un exemple d'utilisation des setoïdes, ce qui simplifie l'écriture des preuves.

1. Introduction

L'évaluation partielle transforme un programme en un programme spécialisé, en fonction de valeurs d'une partie des variables d'entrée. Pour ce faire, l'évaluation partielle exploite les valeurs de variables connues dès la compilation afin d'évaluer symboliquement ce qui est connu dans le programme. Cette évaluation symbolique permet de simplifier certaines expressions et instructions du programme. Par exemple, une conditionnelle peut être simplifiée en l'une de ses branches, cette branche étant elle-même simplifiée. La figure 1 définit plus formellement l'évaluation partielle : étant donnés un programme P et ses variables d'entrée $x_1 \dots x_n, y_1 \dots y_m$, l'évaluation partielle de P pour les valeurs respectives $c_1 \dots c_n$ des $x_1 \dots x_n$ fournit un programme P' qui se comporte comme le programme initial P : quelles que soient les valeurs respectives $d_1 \dots d_m$ des variables $y_1 \dots y_m$, l'exécution de P pour les valeurs $x_1 = c_1 \dots x_n = c_n, y_1 = d_1 \dots y_m = d_m$ et l'exécution de P' pour les valeurs $y_1 = d_1 \dots y_m = d_m$ fournissent les mêmes résultats.

$$P(x_1, \dots x_n, y_1 \dots y_m) \xrightarrow[x_i = c_i, \forall\, i = 1 \dots n]{} P'(y_1 \dots y_m) \quad \textbf{avec} \quad exec(P(c_1, \dots c_n, d_1 \dots d_m)) = exec(P'(d_1 \dots d_m))$$

FIG. 1 – Définition de l'évaluation partielle.

L'évaluation partielle a surtout été utilisée en compilation pour générer des compilateurs à partir d'interprètes ou encore pour générer du code efficace (dans différents domaines tels que l'optimisation de programmes numériques, le graphisme, les systèmes d'exploitation). De nombreux travaux existent sur les langages fonctionnels ; il en existe aussi pour les langages logiques et les langages impératifs. Les références [1, 2, 3, 4] constituent un panorama assez complet de ces travaux.

Nous avons utilisé et adapté l'évaluation partielle dans le cadre de l'aide à la maintenance de programmes impératifs pour améliorer la compréhension de ces derniers [5, 6, 7, 8]. Notre démarche

a été expérimentée sur des applications scientifiques industrielles écrites en Fortran. Ces applications sont de grande taille (plusieurs milliers de lignes d'instructions) et ont connu de nombreuses évolutions sur plusieurs années. Il est aujourd'hui devenu très difficile de maintenir ces applications. Par exemple, modifier la valeur d'une variable en un point de programme prend plusieurs semaines à une personne ne connaissant pas l'application.

Notre démarche a permis de réduire sensiblement le code à comprendre car dans les programmes générés par notre approche, beaucoup de variables auxiliaires qui ne sont pas significatives pour les personnes en charge de la maintenance de l'application ont disparu. De plus, ces programmes contiennent beaucoup moins d'instructions que les programmes initiaux. En effet, dans les applications que nous avons étudiées, de nombreuses conditionnelles ont été remplacées par l'une de leurs branches, et cette branche a elle-même été simplifiée.

Notre démarche a fourni de bons résultats sur les applications que nous avons étudiées, mais elle ne peut être appliquée à n'importe quelle application. Les applications que nous avons étudiées sont un bon candidat pour notre démarche car la seule connaissance commune à toutes les personnes en charge de la maintenance s'exprimait précisément à l'aide d'égalités entre des variables d'entrée et des valeurs. Cette connaissance commune se traduit au niveau des programmes par de nombreuses alternatives aiguillant sur différentes variantes d'un même algorithme en fonction des valeurs de ces mêmes variables d'entrée. Par exemple, les applications que nous avons étudiées modélisent des écoulements de fluides dans différents contextes (par exemple écoulement d'un fluide contenant un polluant particulier à travers un volume pouvant être poreux avec un coefficient particulier de porosité), et ces contextes sont décrits par des égalités entre paramètres (par exemple un indice de porosité) et valeurs. [8] caractérise les applications que nous avons étudiées et montre comment nous avons appliqué l'évaluation partielle.

Notre évaluateur partiel transformant des programmes opérationnels, il est essentiel dans notre démarche d'être sûr de fournir un logiciel de confiance n'introduisant pas d'erreur supplémentaire dans ces programmes. C'est pourquoi nous avons suivi le processus de développement suivant [9] :

- spécification d'une sémantique dynamique d'un langage impératif,
- partant de cette sémantique dynamique, spécification d'une évaluation partielle,
- preuve de correction de la transformation de programmes par rapport à la sémantique dynamique,
- génération d'un prototype.

Nous avons utilisé Centaur [10] pour spécifier en sémantique naturelle [11] la sémantique dynamique et l'évaluation partielle de programmes impératifs et générer ensuite automatiquement un prototype en Prolog. Plus précisément, une première spécification a permis de définir un mini-langage impératif très général (noté *Imp*) ainsi qu'un évaluateur partiel pour ce langage. La correction sémantique de l'évaluation partielle a été prouvée à la main [9]. Dans le cadre d'une étude de cas d'un couplage entre Centaur et Coq, Bertot et Fraer ont vérifié avec Coq la correction sémantique de certaines des transformations élémentaires effectuées par notre évaluation partielle en dehors du contexte de l'évaluation partielle [12]. Nous avons ensuite défini pour Fortran 90 [13] une deuxième spécification plus complète, afin de prendre en compte les appels de sous-programmes et les effets de bords dus aux pointeurs. La preuve de correction sémantique a été faite entièrement à la main pour cette extension [8].

Traditionnellement, il existe deux stratégies d'évaluation partielle : l'évaluation partielle "on-line" qui se déroule en une seule étape et l'évaluation partielle "off-line" qui effectue d'abord une analyse de temps de liaison avant de spécialiser un programme. Un évaluateur partiel "on-line" peut être vu comme un interprète non standard, alors qu'un évaluateur partiel "off-line" procède à l'image d'un compilateur. L'évaluation partielle "on-line", une analyse plus précise que l'évaluation partielle "off-line", est beaucoup plus coûteuse que cette dernière. C'est pourquoi la stratégie principalement étudiée est l'évaluation partielle "off-line". Notre objectif étant de faciliter la compréhension du code et non pas de produire un code qui soit le plus efficace possible, nous avons choisi une évaluation partielle "on-line". Une raison supplémentaire en faveur de ce choix est que notre analyse ne déplie pas les

instructions, contrairement à ce qui est fait usuellement en évaluation partielle, afin de produire un programme le plus efficace possible (c'est-à-dire contenant un maximum d'instructions de bas niveau). [5] détaille notre statégie et précise les différences avec une évaluation partielle "on-line" plus classique.

Les preuves associées aux évaluateurs partiels concernent la terminaison de l'analyse et surtout la correction de l'analyse de temps de liaison, qui a été très étudiée dans le cadre de l'interprétation abstraite ainsi que dans celui de la théorie des types [14, 15, 16, 17, 18]. Ces preuves sont assez éloignées des notres. Les assistants à la preuve sont encore très peu utilisés pour prouver des évaluations partielles : à notre connaissance, seule une exprérience avec Elf a été tentée afin de spécifier et prouver un évaluateur partiel "off-line" fondé sur une analyse de types, et concernant un λ-calcul simplement typé [19]. Notre évaluation partielle ne dépliant pas d'instruction et se déroulant en une seule étape, nos preuves de correction de l'évaluation partielle par rapport à une sémantique sont beaucoup plus simples. Nos preuves se rapprochent plutôt des preuves de traduction d'un langage en un autre.

Cet article présente une formalisation en Coq version 7.3 [20] de la sémantique dynamique du langage *Imp* ainsi que de notre évaluation partielle. Le système Centaur n'étant couplé à aucun démonstrateur de théorèmes, des travaux [12, 21] ont étudié la traduction automatique d'une spécification en sémantique naturelle exprimée en Typol (la sémantique disponible dans Centaur) en une spécification équivalente en Coq. Cependant, ces travaux n'ont pas été poursuivis. De plus, notre but n'est pas d'adapter à peu de frais la spécification que nous avions écrite en sémantique naturelle dans Centaur. Ceci aurait fourni une spécification Coq de trop bas niveau. Au contraire, nous souhaitons spécifier de façon la plus abstraite possible le langage que nous analysons ainsi que notre évaluation partielle de programmes impératifs.

Par exemple, nous avons choisi de modéliser les états-mémoire ainsi que les opérations associées à un haut niveau d'abstraction. Nous avons défini une notion générale de table d'association, ainsi qu'une relation d'inclusion et une relation d'équivalence entre tables. Cette formalisation à un haut niveau d'abstraction a nécessité le développement en Coq d'une couche de plus bas niveau définissant plus généralement la notion de table générique ainsi que des opérations de mises à jour de ces tables. Nous avons modélisé les états-mémoire comme étant des tables particulières. Ainsi, l'évaluation partielle n'utilise que les opérations de haut niveau que nous avons définies sur les tables. Enfin, la correction de l'évaluation partielle par rapport à la sémantique dynamique a été prouvée en Coq.

Cet article est organisé de la façon suivante. La section 2 présente notre modélisation Coq des états-mémoire. Ensuite, la section 3 précise la syntaxe abstraite et la sémantique dynamique de notre langage impératif. Puis, la section 4 détaille notre évaluation partielle. Enfin, la section 5 est consacrée à la preuve de correction de l'évaluation partielle par rapport à la sémantique ayant été définie.

2. Modélisation des états-mémoire

Formaliser une analyse de programmes impératifs nécessite de formaliser la notion d'état-mémoire, qui permet de conserver en tout point de programme les valeurs connues des variables. Cette section présente d'abord notre formalisation des états-mémoire. Elle détaille ensuite quelques propriétés des états-mémoire. Enfin, une formalisation s'appuyant sur la notion de setoïde est introduite.

2.1. Définition des tables

Un état-mémoire est une fonction partielle définie sur les variables, à valeurs dans l'ensemble des valeurs possibles. En effet, en tout point de programme il peut exister des variables dont la valeur n'a pas encore été calculée. Les deux représentations Coq les plus communes des états-mémoire utilisent soit une liste de couples (variable, valeur), soit une fonction totale. Ces représentations ne nous conviennent pas. En effet, d'une part, représenter un état-mémoire par une liste de couples nécessite de gérer l'unicité des variables, puisque toute variable possède au plus une valeur. D'autre part,

représenter un état-mémoire par une fonction totale nécessite l'introduction d'une valeur particulière $\perp$ pour les variables dont les valeurs ne sont pas connues, et donc la manipulation explicite de cette valeur $\perp$.

Afin de tirer parti de la fonctionnalité de Coq, nous avons donc représenté un état-mémoire par la notion de table définie dans [22] et considéré des tables dont les clés sont les variables et les informations les valeurs connues au point de programme courant. Ce choix a simplifié la preuve de correction sémantique de l'évaluation partielle. Il a par exemple simplifié les preuves concernant le déterminisme de l'évaluation des expressions et aussi de l'exécution des instructions, en sémantique dynamique comme en évaluation partielle.

[22] définit la notion de table générique paramétrée par les types des clés (A) et des informations associées (B). Une table est un couple formé du domaine de la fonction partielle sous-jacente et d'une fonction totale de type $A \rightarrow B$. L'accès à la table, c'est-à-dire à la fonction encapsulée, est conditionné par la vérification que l'élément clé utilisé appartient au domaine de la table : *(apply_table t a b)* signifie que $a \in dom(t)$ et $b = t(a)$ (avec quelques abus de notations). Les définitions du type *table* et du prédicat *apply_table* sont données à la figure 2.

```
Variable A : Set.
Inductive list : Set :=Nil : list | Cons : A → list → list.
Variable B : Set.

Inductive table : Set := intro_table : list → (A → B) → table.

Inductive apply_table : table → A → B → Prop := intro_apply_table : (l :A)(dom :list)(f :A→B)
        (in_prop l dom)→ (apply_table (intro_table dom f) l (f l)).
```

FIG. 2 – Définition d'une table.

Dans notre développement formel nous avons utilisé le type *table* comme un type abstrait algébrique, c'est-à-dire un type auquel sont associés des lemmes relatifs aux tables. Pour les besoins de l'évaluation partielle, nous avons défini de nouvelles opérations sur les tables (suppression d'un élément du domaine, intersection de deux tables) et de nouveaux lemmes prouvant des propriétés sur ces opérations, ainsi qu'une relation d'inclusion entre tables et une relation d'équivalence. La figure 3 détaille les relations d'inclusion et d'équivalence ainsi que l'intersection de deux tables.

La relation d'inclusion *inclus* est définie à partir du prédicat *apply_table* : une table t est incluse dans une table t' si et seulement si le domaine de t est inclus dans le domaine de t' et si sur le domaine de t elles contiennent les mêmes informations. La relation d'équivalence *equiv* est définie à partir de la relation d'inclusion *inclus* : deux tables sont équivalentes si chacune est incluse dans l'autre. Les propriétés de réflexivité, symétrie et transitivité de la relation d'équivalence ont été prouvées en utilisant uniquement des lemmes sur *apply_table* déjà prouvés dans [22]. L'intersection de deux tables est définie inductivement par un seul constructeur *inter_intro*. La définition de la figure 3 exprime que l'intersection de deux tables t et t' est la table t'' constituée des paires *(a,b)* appartenant aux deux tables t et t'. La table t'' possède de plus la propriété d'être incluse dans chacune des deux tables t et t'.

Les opérations ont été spécifiées à un haut niveau d'abstraction car elles ont été définies en fonction de l'opération de plus bas niveau *apply_table*, et non pas à partir d'opérations élémentaires manipulant directement les composants des tables. Les opérations de différence et d'union ont également été spécifiées ainsi. N'étant pas utilisées dans le développement présenté dans cet article, elles ne sont pas détaillées dans la figure 3.

La figure 4 montre quelques lemmes définis sur les tables. Le premier, *apply_table_det*, exprime que les tables sont des fonctions : si deux couples $(a, b1)$ et $(a, b2)$ appartiennent à une même table, alors nécessairement $b1 = b2$. Le suivant, *inclus_trans* exprime la transitivité de l'inclusion. Le troisième,

```
Definition inclus [t,t' :table] : Prop := (a :A)(b :B) (apply_table t a b) → (apply_table t' a b).

Definition equiv [t,t' :table] : Prop := (inclus t' t) ∧ (inclus t t').

Inductive table_inter : (table A B) → (table A B) → (table A B) → Prop :=
 inter_intro : (t,t',t" :(table A B))
               ( (a :A) (b :B)
                     (apply_table t a b) → (apply_table t' a b) →
                     (apply_table t" a b) ) →
               (inclus t" t) → (inclus t" t') →
               (table_inter t t' t").
```

FIG. 3 – Définition des opérations de haut niveau d'accès aux tables.

equiv_inclus exprime que la relation d'équivalence conserve l'inclusion entre deux tables. Les trois derniers (*eqrefl*, *eqsym* et *eqtrans*) expriment les propriétés de réflexivité, symétrie et transitivité de la relation d'équivalence. La transitivité de la relation d'équivalence se prouve à l'aide la la transitivité de la relation d'inclusion.

```
Lemma apply_table_det : (t :table) (a :A)(b1,b2 :B)
  (apply_table t a b1) → (apply_table t a b2) → b1=b2.

Lemma inclus_trans : (t,t',t" :table)
  (inclus t t') → (inclus t' t") → (inclus t t").

Lemma equiv_inclus : (t,t',t" :table)
   (inclus t t') → (equiv t" t') → (inclus t t").

Lemma eqrefl : (t :table)   (equiv t t).

Lemma eqsym : (t,t' :table)   (equiv t t')→ (equiv t' t).

Lemma eqtrans : (t,t',t" :table)   (equiv t t') → (equiv t' t") → (equiv t t").
```

FIG. 4 – Quelques théorèmes relatifs aux opérations sur les tables.

2.2. Un setoïde pour les états-mémoire

Au vu de notre définition des tables et des opérations associées, il est intéressant de considérer l'ensemble des tables comme un setoïde. En effet, un setoïde est un type muni d'une relation d'équivalence. Or, pour les besoins de l'évaluation partielle, nous avons dû définir une notion d'équivalence entre deux tables pour signifier que deux états-mémoire contiennent exactement les mêmes informations, sans pour autant être identiques. Cette différence résulte des ajouts et suppressions de couples pouvant se produire en tout point de programme.

L'intérêt d'utiliser des setoïdes dans notre modélisation réside dans l'utilisation de l'équivalence comme une égalité. En effet, considérer des tables équivalentes plutôt qu'une seule table "pollue" les preuves. Par exemple, lors de la preuve de la correction sémantique de l'instruction séquence, il est nécessaire d'introduire des états-mémoire intermédiaires relatifs au point de programme situé après l'exécution de la première instruction, et ceci pour la sémantique dynamique comme pour l'évaluation partielle. On obtient ainsi différents états-mémoire équivalents, ce qui alourdit la preuve. Un autre exemple est donné par le théorème *equiv_sem* de la figure 9 qui contient un quantificateur existentiel. Pour prouver ce théorème, il est nécessaire d'exhiber différents états-mémoire équivalents.

D'où l'intérêt d'utiliser des setoïdes et des morphismes de setoïdes (c'est-à-dire des fonctions entre setoïdes respectant la structure de setoïdes) afin de raisonner globalement sur un état-mémoire qui sera représenté par différentes tables équivalentes.

La dernière version de Coq permet de définir des setoïdes, puis de faire ensuite de la réécriture entre éléments équivalents comme cela se fait sur des termes égaux [23]. Une fois définies les tables, leurs opérations associées et la relation d'équivalence, nous avons défini la structure de setoïde : la figure 5 montre la définition en Coq du setoïde *tableseto* relatif au type *table* et à la relation d'équivalence *equiv*. Cette définition suppose que les propriétés de réflexivité (cf. le lemme *eqrefl* de la figure 4), symétrie (*eqsym*) et transitivité (*eqtrans*) de la relation d'équivalence ont été préalablement prouvées.

```
Definition tableseto : (Setoid_Theory table equiv).
Apply Build_Setoid_Theory.                    (* preuve qu'equiv est une relation d'équivalence *)
Intros x ; Apply eqrefl.
Intros x y Heq ; Apply eqsym ; Assumption.
Intros x y z Heq1 Heq2 ; Apply (eqtrans Heq1 Heq2) ; Assumption.
Qed.
```

FIG. 5 – Définition du setoïde relatif au type *table*.

Le setoïde *tableseto* étant défini, lorsque deux tables sont equivalentes, chacune d'entre elles peut être remplacée par l'autre quand celles-ci sont arguments de morphismes de setoïdes. Nous avons par exemple montré que les deux opérations *inclus* et *apply_table* sont de tels morphismes. La figure 6 montre l'utilisation de ces notions sur un exemple : puisque *inclus* est un morphisme de setoïdes, si deux tables *t'* et *t"* sont équivalentes (c'est-à-dire *(equiv t" t')*), alors la tactique Coq *Setoid_replace* remplace l'une d'entre elles (par exemple *t"*) par l'autre dans le but *(inclus t t")*. Ce dernier devient donc *(inclus t t')*, qui est une hypothèse. L'utilisation de cette tactique Coq a ainsi simplifié la preuve de nos théorèmes.

```
Theorem equiv_inclus : (t,t',t" :table)
    (inclus t t') → (equiv t" t') → (inclus t t").
Proof.
Intros t t' t" Hinclus Hequiv.
Setoid_replace t" with t'.      (* car t" et t' sont des setoïdes et inclus est un morphisme *)
Assumption.                     (* le but à prouver est l'hypothèse d'équivalence Hequiv *)
Qed.
```

FIG. 6 – Utilisation du setoïde relatif au type *table*.

3. Sémantique

Cette section définit d'abord la sémantique de notre langage impératif. Elle précise ensuite certaines propriétés de cette sémantique. Ces propriétés ont été utilisées pour prouver la correction sémantique de l'évaluation partielle. Nous avons prouvé en Coq ces propriétés, mais nous ne détaillerons pas ces preuves dans cet article.

3.1. Définitions de la syntaxe et de la sémantique

Le langage *Imp* est un langage impératif simple. Il peut être vu comme un noyau commun aux langages impératifs, et il est présenté avec davantage de détails dans [24]. Cette section formalise en

sémantique naturelle la sémantique dynamique du langage *Imp*. Les règles de sémantique naturelle sont écrites directement en Coq.

La figure 7 précise en Coq la syntaxe abstraite du langage *Imp*. Les catégories syntaxiques sont les constantes (entières ou booléennes), les variables, les opérateurs apparaissant dans les expressions, les expressions et les instructions. Par souci d'abstraction, les différents opérateurs possibles ne sont pas détaillés (ils le sont dans [24]) ; seules sont considérées les notions d'opérateur unaire et d'opérateur binaire. Vu la similarité du traitement des opérateurs unaires et binaires et pour ne pas surcharger l'article, seule la notion d'opérateur binaire est présentée, bien que la notion d'opérateur unaire ait été également prise en compte dans le travail effectué. Dans cet article, nous avons de plus choisi d'omettre le constructeur *InjBool* lorsque nous mentionnons une valeur booléenne. Par souci de clarté, nous écrivons donc (*Const true*) et non pas (*Const InjBool true*) pour désigner la valeur booléenne correspondant à *true*. Les instructions possibles sont l'instruction vide, l'affectation, la séquence, la conditionnelle et la boucle.

La figure 7 détaille les définitions inductives des expressions et des instructions. Le type des variables est laissé abstrait, il est supposé muni d'une égalité décidable, d'où l'axiome *eq_var_dec*. Les opérateurs binaires sont abstraits, le prédicat *op_bin* définit leur interprétation, nous supposons de plus qu'ils sont déterministes (d'où l'axiome *op_bin_det*). Un programme est une instruction, généralement une séquence. Le type des états-mémoire utilise le type générique table avec les paramètres *var* et *constante*.

```
Inductive constante : Set := InjNat : nat → constante | InjBool : bool → constante.
Variable var, op : Set.
Variable op_bin : op → constante → constante → constante →Prop

Hypothesis eq_var_dec : (x,y :var) {x=y}+{¬x=y}.

Inductive exp : Set :=
    Const : constante → exp
  | Var : var → exp
  | OpBin : op → exp → exp → exp.

Axiom op_bin_det : (x,y,z,t :constante) (o :op)
      (op_bin o x y z) → (op_bin o x y t) →z =t.

Inductive inst : Set :=
    Skip : inst
  | Aff : var → exp → inst
  | Seq : inst → inst → inst
  | If : exp → inst → inst → inst
  | While : exp → inst → inst.

Definition programme := inst.

Definition etat := (table var constante).
```

FIG. 7 – Syntaxe abstraite du langage *Imp*.

La figure 8 précise la sémantique du langage *Imp*. Celle-ci se compose de deux prédicats inductifs, *eval* et *exec*. Étant donné un état-mémoire :

- l'évaluation d'une expression fournit une constante, que l'expression soit une constante (constructeur *eval_const*), une variable (constructeur *eval_ident*) ou composée d'opérateurs et d'opérandes (constructeur *eval_op*),
- l'exécution d'une instruction fournit un nouvel état-mémoire.

```
Inductive eval : etat → exp → constante →Prop :=
          eval_const : (m :etat) (v :constante)    (eval m (Const  v) v)
        | eval_ident : (m :etat) (id :var) (v :constante)
              (apply_table m id v) →
              (eval m (Var id) v)
        | eval_op : (e1, e2 :exp) (m :etat) (v1,v2, v :constante) (o :op)
              (eval m e1 v1) → (eval m e2 v2) → (op_bin o v1 v2 v) →
              (eval m (OpBin o e1 e2) v).

Inductive exec : etat → inst → etat → Prop :=
   exec_skip : (m :etat) (exec m Skip m)
| exec_aff : (m :etat)(id :var)(e :exp)(v :constante)
                       (eval m e v)→
                       (exec m (Aff id e) (add_table id v m))
| exec_seq : (m,m1,m2 :etat) (i1,i2 :inst)
                      (exec m i1 m1)→(exec m1 i2 m2) →
                      (exec m (Seq i1 i2) m2)
| exec_if_t : (m,m1 :etat) (e :exp) (i1,i2 :inst)
        (eval m e true) → (exec m i1 m1) →
        (exec m (If e i1 i2) m1)
| exec_if_f : (m,m2 :etat) (e :exp) (i1,i2 :inst)
        (eval m e false) → (exec m i2 m2) →
        (exec m (If e i1 i2) m2)
| exec_while_t : (i :inst) (e :exp) (m,m1,m2 :etat)
        (eval m e true) → (exec m i m1) → (exec m1 (While e i) m2) →
        (exec m (While e i) m2)
| exec_while_f : (i :inst) (e :exp) (m :etat)
        (eval m e false) →
        (exec m (While e i) m).
```

FIG. 8 – Sémantique opérationnelle du langage *Imp*.

3.2. Propriétés de la sémantique

Les propriétés de la sémantique ayant été utilisées lors de la preuve de correction de l'évaluation partielle sont données à la figure 9. Les deux premières établissent que la sémantique est déterministe, puisque l'évaluation des expressions et l'exécution des instructions le sont (théorèmes *eval_det* et *exec_det*). La troisième propriété *equiv_eval* exprime que si deux états-mémoire sont équivalents, l'évaluation de toute expression fournit la même valeur dans les deux états-mémoire. Cette propriété est obtenue gratuitement grâce à la définition du setoïde des tables. Enfin, la dernière propriété *equiv_sem* ressemble à la précédente et s'applique à l'exécution des instructions : étant donnés deux états-mémoire équivalents *m* et *m'*, si l'état-mémoire *m"* résultant de l'exécution d'une instruction dans *m'* existe, alors l'état-mémoire *m"'* résultant de l'exécution de l'instruction dans *m* existe également et il est équivalent à *m"*.

4. Évaluation partielle

L'évaluation partielle exécute symboliquement un programme en ne prenant en compte que certaines valeurs des variables d'entrées. Aussi, en tout point de programme, l'évaluation partielle d'une expression fournit soit une constante, soit une expression résiduelle dans laquelle apparaissent

```
Theorem eval_det : (e :exp) (m :etat) (v,v' : constante)
  (eval m e v) → (eval m e v') → v=v'.

Theorem exec_det : (i :inst)(m,m',m" :etat)
  (exec m i m') → (exec m i m") → m'=m".

Theorem equiv_eval : (e :exp)(m,m' :etat)(v :constante)
      (equiv m m') → (eval m e v) → (eval m' e v).

Theorem equiv_sem : (i :inst) (m,m',m" :etat)
      (exec m' i m") → (equiv m m') →
      (∃ m"' :etat | (equiv m" m"') ∧ (exec m i m"')).
```

FIG. 9 – Théorèmes portant sur la sémantique du langage et nécessaires à la preuve de correction.

```
Inductive op_ep_bin :op → exp → exp → exp →Prop :=
      op_bin_ep_def : (o :op) (x1,x2,x3 :constante)
      (op_bin o x1 x2 x3) →
      (op_ep_bin o (Const x1) (Const x2) (Const x3))
    | op_bin_ep : (e1,e2 :exp )(o :op)
      (not_val e1) ∨ (not_val e2) →
      (op_ep_bin o e1 e2 (OpBin o e1 e2)).

Inductive exp_ep : etat → exp → exp → Prop :=
         exp_ep_const : (m :etat) (v :constante)
           (exp_ep m (Const v) (Const v))
        | exp_ep_ident_in : (m :etat) (id :var) (v :constante)
            (apply_table m id v) → (exp_ep m (Var id) (Const v))
        | exp_ep_ident_not_in : (m :etat) (id :var)
            ¬(in_dom id m) → (exp_ep m (Var id) (Var id))
        | ep_bin :(e1,e2,e'1,e'2,e' :exp) (m :etat) (o :op)
            (exp_ep m e1 e'1) → (exp_ep m e2 e'2) → (op_ep_bin o e'1 e'2 t)→
            (exp_ep m (OpBin o e1 e2) e').
```

FIG. 10 – Définition de l'évaluation partielle des expressions.

des variables dont la valeur n'est pas connue au point de programme considéré. Par exemple, sachant que y vaut 1, $x + y$ s'évalue partiellement en 3 si la valeur de x est 2 et en $x + 1$ si la valeur de x n'est pas connue. Aussi, l'évaluation partielle d'une expression soit évalue totalement l'expression, soit ne l'évalue que partiellement. L'évaluation partielle d'une expression renvoie donc une expression et non plus seulement une constante comme le faisait la sémantique dynamique. L'évaluation partielle d'une expression est définie dans la figure 10. Elle se compose de deux prédicats inductifs *op_ep_bin* et *exp_ep*. L'évaluation partielle d'une expression comprend le cas d'évaluation totale de l'expression (c'est-à-dire le cas correspondant à la sémantique dynamique) plus les cas d'évaluation strictement partielle de l'expression. L'évaluation totale d'une expression est représentée par les constructeurs *exp_ep_const*, *exp_ep_ident_in* et dans le cas où le constructeur *op_bin_ep_def* du prédicat *op_ep_bin* est appliqué, par le constructeur *ep_bin*. L'évaluation strictement partielle d'une expression correspond soit à l'évaluation partielle d'une variable dont la valeur est inconnue, soit à l'évaluation partielle d'une expresssion dont un opérande au moins n'est pas évalué complètement. L'expression résiduelle est alors construite à partir des opérandes ayant été partiellement évalués et de l'opérateur de l'expression initiale (constructeur *op_bin_ep* appelé par le constructeur *ep_bin*).

```
Inductive inst_ep : etat → inst → inst → etat → Prop :=
  inst_ep_skip : (m :etat) (inst_ep m Skip Skip m)
| inst_ep_aff_total : (m,m' :etat) (v :var) (e :exp) (n :constante)
      (exp_ep m e (Const n))→
      (inst_ep m (Aff v e) (Aff v (Const n)) (add_table v n m))
| inst_ep_aff_partiel : (m,m',m" :etat) (v :var) (e,e' :exp)
      (exp_ep m e e') → (not_val e') → (remove_from_table m v m") →
      (inst_ep m (Aff v e) (Aff v e') m")
| inst_ep_seq : (m,m1,m2 :etat) (i1,i2,i'1, i'2 :inst)
      (inst_ep m i1 i'1 m1) → (inst_ep m1 i2 i'2 m2) →
      (inst_ep m (Seq i1 i2) (Seq i'1 i'2) m2)
| inst_ep_if_t : (m,m' :etat) (e :exp) (i1,i2,i'1 :inst)
      (exp_ep m e (Const true)) → (inst_ep m i1 i'1 m') →
      (inst_ep m (If e i1 i2) i'1 m')
| inst_ep_if_f : (m,m' :etat) (e :exp) (i1,i2,i'2 :inst)
      (exp_ep m e (Const false)) → (inst_ep m i2 i'2 m') →
      (inst_ep m (If e i1 i2) i'2 m')
| inst_ep_if_partiel : (m,m1,m2,m3 :etat) (e,e' :exp) (i1,i2,i'1,i'2 :inst)
      (exp_ep m e e') → (not_bool e') → (inst_ep m i1 i'1 m1) →
      (inst_ep m i2 i'2 m2) → (table_inter m1 m2 m3) →
      (inst_ep m (If e i1 i2) (If e' i'1 i'2) m3)
| inst_ep_while_f : (m :etat) (e :exp) (i :inst)
      (exp_ep m e (Const false)) → (inst_ep m (While e i) Skip m)
| inst_ep_while_not_f : (m,m' :etat) (e,e' :exp ) (i,i' :inst)
      (calcul_inv m i m') → (exp_ep m' e e') → (not_false e') →
      (inst_ep m' i i' m') → (inst_ep m (While e i) (While e' i') m').
```

FIG. 11 – Définition de l'évaluation partielle des instructions.

De même, la définition de l'évaluation partielle d'une instruction reprend la définition de la sémantique dynamique de l'instruction et rajoute les cas d'évaluation strictement partielle, comme le montre la figure 11. Chacun de ces cas est représenté par un constructeur dans la définition de l'évaluation partielle :

- constructeur *inst_ep_aff_partiel* dans le cas d'une affectation dont l'expression en partie droite a une valeur inconnue,
- constructeur *inst_ep_if_partiel* dans le cas d'un conditionnelle dont la condition a une valeur inconnue.

Ces deux constructeurs correspondent aux cas où une expression contient au moins une variable dont la valeur n'est pas connue au point de programme courant (cette variable n'appartient pas au domaine de l'état-mémoire).

Alors que la sémantique dynamique ne détecte principalement que des relations entre variables et valeurs, l'évaluation partielle détecte plus généralement des relations entre variables et expressions. Mais, ces relations ne sont pas toutes exploitées puisque l'évaluation partielle propage à travers le programme analysé uniquement les valeurs statiquement connues des variables. Aussi, nous utilisons pour l'évaluation partielle la notion d'état-mémoire que nous avons définie pour la sémantique dynamique.

Étant donné un état-mémoire m, lorsque dans m l'expression e ne s'évalue pas totalement (c'est-à-dire s'évalue en une expression e' non constante) :

- l'affectation $v := e$ est simplifiée en l'affectation $v := e'$,

– au point de programme suivant l'affectation, dans m, la valeur de la variable v devient inconnue, quelle que soit son ancienne valeur éventuelle. Autrement formulé, la variable v est supprimée de m, si elle existait.

Étant donné un état-mémoire m, lorsque dans m l'expression e ne s'évalue pas totalement (c'est-à-dire s'évalue en *e'* qui est une expression non constante) :

– la conditionnnelle *if e then i1 else i2* est simplifiée en la conditionnelle *if e' then i'1 else i'2*, la branche *i'1* (resp. *i'2*) résultant de l'évaluation partielle de la branche *i1* (resp. *i2*) en fonction des valeurs de m,
– l'état-mémoire résultant est l'intersection des deux états-mémoire calculés lors de l'évaluation partielle de chaque branche de la conditionnelle. Autrement formulé, à l'issue de l'évaluation partielle des deux branches, l'état-mémoire est constitué uniquement des variables dont la valeur est connue et identique dans les deux branches.

L'évaluation partielle d'une boucle soit supprime une boucle connue comme non exécutée (constructeur *inst_ep_while_f*), soit la transforme en une boucle simplifiée (constructeur *inst_ep_while_not_f*). Il en est d'ailleurs ainsi pour les conditionnelles, qui sont soit supprimées (c'est-à-dire réduites en une de leurs branches au plus), soit simplifiées en des conditionnelles dont chaque branche est simplifiée. Cette transformation repose sur le calcul d'un "invariant de boucle". La boucle n'est jamais dépliée, contrairement à ce qui se fait en évaluation partielle classique.

L'invariant de boucle, spécifié par le prédicat *calcul_inv* est un état-mémoire qui contient les variables qui ne sont pas modifiées par les instructions du corps de la boucle. Cet invariant est "simple", puisqu'il ne contient pas de variable définie mais seulement des variables utilisées dans le corps de la boucle. Le calcul de l'invariant n'est pas détaillé ici pour ne pas surcharger l'article. Étant donné un état-mémoire m, lorsque dans m l'expression e ne s'évalue pas à *false*, alors un invariant de boucle *m'* est calculé. L'expression e et le corps i de la boucle sont alors évalués partiellement dans *m'* en respectivement *e'* et *i'*. La boucle initiale est ainsi simplifiée en une boucle de condition *e'* et de corps *i'*.

5. Preuve de correction

Le but de cette partie est de montrer que la transformation de programmes que nous venons de définir est sémantiquement correcte. À l'issue de l'exécution d'un programme P conformément à la sémantique dynamique présentée précédemment, l'état-mémoire m_f contient les valeurs finales des variables. Il s'agit de montrer qu'étant données des valeurs particulières pour certaines variables d'entrée, le programme initial P et son programme *P'* spécialisé pour ces valeurs fournissent les mêmes résultats pour des valeurs identiques des variables d'entrée, donc un état-mémoire m'_f équivalent à m_f. Remarquons que notre évaluation partielle ne dépliant pas d'instruction, le problème de terminaison de l'exécution de *P'* ne se pose pas.

Nous avons utilisé le schéma général défini dans [25] pour montrer la correction sémantique de l'évaluation partielle. Dans notre cas, il s'agit de montrer que le diagramme de la figure 12 commute. La commutativité du diagramme exprime cette propriété : partant de P, il existe deux chemins permettant d'obtenir le résultat : soit directement, soit en passant par *P'*. Plus précisément, cette propriété se démontre à l'aide de deux théorèmes. Le premier exprime qu'étant données des valeurs finales obtenues en suivant les flèches 1 et 2, alors ces valeurs sont également obtenues en suivant les flèches 3 et 4. Le second théorème est le théorème réciproque du premier.

Ces deux théorèmes sont ennoncés à la figure 13. Le premier théorème, *inst_completeness*, exprime la complétude de l'évaluation partielle : tout résultat fourni par l'éxécution du programme initial est aussi le résultat fourni par l'exécution du programme spécialisé. Le second théorème, *inst_soundness*, exprime la cohérence de l'évaluation partielle : tout résultat fourni par l'exécution du programme spécialisé est aussi fourni par l'exécution du programme initial. Plus précisément, comme les deux

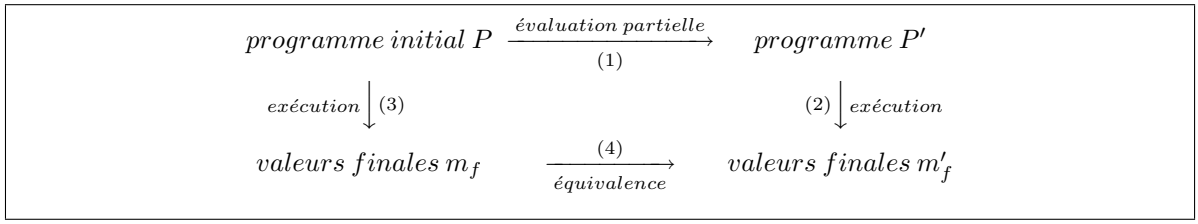

FIG. 12 – Diagramme traduisant la preuve de correction sémantique de l'évaluation partielle.

Theorem *inst_completeness* : (*i,i'* :*inst*) (*m0,m,m',m",m"'* :*etat*)
(*inst_ep m0 i i' m'*) → (*inclus m0 m*) → (*exec m i m"*) →
(*exec m i' m"'*) → (*equiv m" m"'*).

Theorem *inst_soundness* : (*i,i'* : *inst*) (*m0,m,m',m",m"'* :*etat*)
(*inst_ep m0 i i' m'*) → (*inclus m0 m*) → (*exec m i' m"*) →
(*exec m i m"'*) → (*equiv m" m"'*).

FIG. 13 – Correction de l'évaluation partielle.

programmes sont différents (du fait des simplifications introduites par l'évaluation partielle), les deux états-mémoire contenant les valeurs finales des variables des deux programmes sont équivalents : ils contiennent les mêmes variables et la valeur de chaque variable est la même dans chaque état-mémoire (cf. la définition de la relation d'équivalence entre tables de la figure 3). Chacun de ces deux théorèmes se prouve par induction de règles utilisées au cours de la preuve des théorèmes de la figure 12.

La figure 14 détaille les théorèmes les plus intéressants. D'autres propriétés de moindre intérêt ont été prouvées mais elles ne sont pas présentées dans cet article. [1] L'ensemble des propriétés de la figure 14 montre que la sémantique dynamique est un cas particulier de l'évaluation partielle. Les premières propriétés concernent l'évaluation des expressions. La première propriété, *eval_ep_eval*, exprime que pour toute expression et tout état-mémoire permettant de connaître la valeur de l'expression, l'évaluation de l'expression par la sémantique dynamique et l'évaluation de l'expression par l'évaluation partielle fournissent une même valeur. Une propriété similaire mais relative aux opérateurs binaires a été démontrée. La deuxième propriété, *plus_large*, exprime que pour un état-mémoire *m0*, si l'évaluation partielle d'une expression a fourni une constante, alors pour un état-mémoire contenant *m0*, c'est-à-dire fixant quelques variables supplémentaires, l'évaluation partielle de cette expression fournira la même constante.

L'évaluation partielle exécute symboliquement un programme en ne considérant que certaines valeurs des variables d'entrée. Dans les trois dernières propriétés de la figure 14, *m0* représente l'état-mémoire initial relatif à l'évaluation partielle et *m* celui relatif à la sémantique dynamique, c'est pourquoi *m0* est inclus dans *m*. La troisième propriété, *exp_ep_correct*, exprime la correction sémantique de l'évaluation partielle des expressions : lorsque l'évaluation partielle d'une expression *e* fournit une expression résiduelle *e'*, si dans *m e* s'évalue à une constante *v*, alors dans *m' e'* vaut également *v*, et réciproquement.

La quatrième propriété *eval_eq_ep* traite le cas où l'évaluation partielle de *e* dans *m0* fournit une constante *v2*. Cette constante est également la valeur *v1* fournie par l'évaluation (totale) de *e* dans *m* lors de la sémantique dynamique. La dernière propriété *ep_inclus_sem* exprime que l'inclusion des états-mémoire est conservée à l'issue du traitement des instructions : pour toute instruction *i*, soit *m"* l'état-mémoire résultant de l'exécution de *i* dans *m* et soit *m'* l'état-mémoire résultant de l'évaluation partielle de *i* dans *m0*, alors *m'* est inclus dans *m"* si *m0* est inclus dans *m*.

[1] L'ensemble du code Coq écrit est accessible depuis la page http ://www.iie.cnam.fr/~blazy

```
Lemma eval_ep_eval : (m :etat)(e :exp)(v :constante)
      (eval m e v) ↔ (exp_ep m e (Const v)).

Lemma plus_large : (m,m' :etat)(e :exp )(n :constante)
      (exp_ep m e (Const n)) → (inclus m m') →
      (exp_ep m' e (Const n)).

Theorem exp_ep_correct : (m0,m :etat) (e,e' :exp) (v :constante)
      (inclus m0 m) → (exp_ep m0 e e') →
      (eval m e v) ↔ (eval m e' v).

Theorem eval_eq_ep : (m0,m :etat) (e :exp) (v1,v2 :constante)
      (inclus m0 m) → (eval m e v1) → (exp_ep m0 e (Const v2)) →
      v1=v2.
Theorem ep_inclus_sem : (i,i' :inst) (m0,m,m',m" :etat)
      (inclus m0 m) → (exec m i m") → (inst_ep m0 i i' m') →
      (inclus m' m").
```

FIG. 14 – Théorèmes portant sur la définition de l'évaluation partielle.

6. Conclusion

Cet article a présenté une formalisation en Coq de la sémantique dynamique d'un mini-langage impératif. Partant de cette formalisation, une transformation de programmes particulière, l'évalaution partielle adaptée pour la compréhension de programmes, a été également formalisée. Ces deux formalisations sont les plus abstraites possible. Elles modélisent les états-mémoire à l'aide de tables génériques que nous avons également présentées dans cet article. La correction sémantique de l'évaluation partielle a ensuite été présentée. Le travail présenté dans cet article représente un peu plus de trois mille lignes de Coq, comprenant soixante-neuf propriétés prouvées, trois axiomes et vingt-et-une définitions de types inductifs. Notre formalisation et nos preuves sont suffisamment générales pour être reprises dans un autre assistant de preuve permettant la définition de types inductifs et possédant des tactiques associées.

Modéliser les tables à l'aide de setoïdes et les opérations sur les tables à l'aide de morphismes n'a pas demandé de travail de preuve supplémentaire, puisque toutes les preuves nécessaires à l'utilisation en Coq de setoïdes et de morphismes auraient dû être faites même sans l'utilisation de setoïde. Aussi, les tables génériques montrent l'intérêt d'utiliser des setoïdes pour spécifier de façon abstraite une analyse de programmes telle que l'évaluation partielle. Cependant, du fait des limitations actuelles dans Coq portant sur la définition des setoïdes (par exemple il n'existe pas de famille de setoïdes) et de leurs morphismes, nous n'avons pas pu définir autant de morphismes que nous l'aurions souhaité. En effet, nous avons découpé nos 3000 lignes de Coq en différents fichiers, ce qui fait que nous aurions eu besoin de pouvoir définir des familles de setoïdes, ainsi que des morphismes répartis dans différents fichiers, ce qui à l'heure actuelle est impossible.

Une autre façon d'internaliser la notion d'équivalence serait d'utiliser la notion de type inductif quotient définie dans [26]. Ceci nécessiterait de manipuler exclusivement des tables normalisées. Il faudrait donc normaliser les tables en chaque point de programme, ce qui alourdirait certainement la preuve de correction sémantique et demanderait de revoir la définition des tables.

Il serait également intéressant d'extraire automatiquement de nos preuves un évaluateur partiel certifié correct. Cela nécessairait de prouver un théorème exprimant que pour tout programme initial P et tout état-mémoire m, il existe un programme final P' et un état-mémoire m' tels que l'évaluation partielle de P étant donné m fournisse m' et P'. L'extraction d'une preuve de cette propriété est en effet une fonction écrite en OCaml fournie par Coq, qui calcule m' et P' étant donnés m et P. La

preuve de cette propriété nécessite de rajouter l'hypothèse que P est correctement typé. Nous avons donc l'intention de spécifier un vérificateur de types pour notre langage.

Enfin, nous comptons poursuivre cette modélisation abstraite en prenant en compte les aspects interprocéduraux ainsi que les effets de bord. Ceci devrait nécessiter de modéliser en Coq d'autres structures de données abstraites, comme nous l'avions fait dans [8]. Nous souhaiterions alors généraliser notre approche à d'autres analyses de programmes que l'évaluation partielle présentée dans cet article. Nous disposerions ainsi d'un environnement générique permettant d'exprimer et de prouver à un haut niveau d'abstraction des propriétés sémantiques de programmes.

Remerciements

L'auteur remercie Catherine Dubois pour les nombreuses discussions, les conseils Coq et la relecture attentive de cet article. L'auteur remercie également Olivier Boite pour l'aide ponctuelle apportée.

Références

[1] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, 1993.

[2] C.Consel and O.Danvy. Tutorial notes on partial evaluation. In *Proc. of Principles Of Progamming Languages Conf.*, pages 493–501, 1993.

[3] O. Danvy, R. Glück, and P. Thiemann, editors. *International seminar on partial evaluation*, volume 1110 of *Lecture Notes in Computer Science*, Dagstuhl castle, February 1996. Springer-Verlag.

[4] ACM. *Symposium on partial evaluation*, number 4 in ACM Computing Surveys, December 1998.

[5] S. Blazy. Partial evaluation for the understanding of Fortran programs. *Journal of Software Engineering and Knowledge Engineering*, 4(4) :535–559, 1994.

[6] S. Blazy and P. Facon. An automatic interprocedural analysis for the understanding of scientific application programs. In Danvy et al. [3], pages 1–16.

[7] S. Blazy and P. Facon. Partial evaluation for program understanding. In [4].

[8] S. Blazy. Specifying and automatically generating a specialization tool for Fortran 90. *Journal of Automated Software Engineering*, 7(4) :345–376, December 2000.

[9] S. Blazy and P. Facon. Formal specification and prototyping of a program specializer. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT '95 : Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 666–680, Aarhus, May 1995. Springer-Verlag.

[10] INRIA. *Centaur 1.2 documentation*, 1994.

[11] G. Kahn. Natural Semantics. In *Proc. of the Symp. on Theoretical Aspects of Comp. Science*, volume 247 of *Lecture Notes in Computer Science*, pages 237–257. Springer-Verlag, 1987.

[12] Y. Bertot and R. Fraer. Reasoning with executable specifications. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT '95 : Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 531–45, Aarhus, May 1995. Springer-Verlag.

[13] ANSI, New York. *Programming language Fortran 90*, 1992. ANSI X3.198-1992 and ISO/IEC 1539-1991(E).

[14] C. Consel and S.-C. Khoo. On-line off-line partial evaluation : Semantic specifications and correctness proofs. *Journal of Functional Programming*, 5(4) :461–500, 1995.

[15] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3) :365–387, July 1993.

[16] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3) :347–363, 1993.

[17] J. Lawall and P. Thiemann. Sound specialization in the presence of computational effects. In *Theoretical Aspects of Computer Software, Sendai, Japan*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190. Springer-Verlag, September 1997.

[18] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5) :507–541, 1997.

[19] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In M. Hermenegildo and S.D. Swierstra, editors, *Proc. of the 7th Int. Symp. on Programming Languages : Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 279–298, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[20] INRIA. *The Coq proof assistant reference manual version 7.3*, May 2002. http ://coq.inria.fr.

[21] D. Terrasse. Encoding natural semantics in Coq. In V. S. Alagar, editor, *Proc. of the Fourth Conference on Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*, pages 230–244, Montreal, Canada, 1995. Springer-Verlag.

[22] O. Boite and C. Dubois. Proving Type Soundness of a Simply Typed ML-like Language with References. In R. Boulton and P. Jackson, editors, *Supplemental Proceedings of TPHOL'01, Informatics Research Report EDI-INF-RR-0046 of University of Edinburgh*, pages 69–84, 2001.

[23] C. Renard. Un peu d'extensionnalité en Coq. Mémoire de DEA, September 2001. Université Paris VI.

[24] G. Winskel. *The formal semantics of programming languages - an introduction*. MIT Press, 1993.

[25] J. Despeyroux. Proof of translation in natural semantics. In *Proc. of the Symposium on Logic in Computer Science*, Cambridge, USA, June 1986.

[26] P. Courtieu. Normalized types. In *CSL*, 2001.