Proving Correctness of the Translation from Mini-ML to the CAM with the Coq Proof Development System

Samuel Boutin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Preuve de correction de la compilation de Mini-ML en code CAM dans le système d'aide à la démonstration COQ

Samuel Boutin

N ° 2536

Avril 1995

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

ISSN 0249-6399

INRIA
ROCQUENCOURT

Preuve de correction de la compilation de Mini-ML en code CAM dans le système d'aide à la démonstration COQ

Samuel Boutin*

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Coq

Rapport de recherche n ° 2536 — Avril 1995 — 34 pages

Résumé : Nous décrivons dans ce papier comment, en utilisant le système d'aide à la démonstration Coq, nous avons réalisé une preuve formelle de la correction d'un compilateur d'un petit langage applicatif contenant des définitions récursives (Mini-ML) en code CAM (Categorical Abstract Machine). Notre objectif a été de mécaniser une preuve présentée dans l'article de J. Despeyroux [9] et écrite à l'aide du langage Typol. Nous utilisons des sémantiques naturelles pour modéliser l'évaluation de nos langages. Nous ne sommes parvenus que partiellement à mécaniser cette preuve de correction. En effet, les spécifications naturelles des langages source et cible contiennent des termes rationnels difficiles à axiomatiser dans l'état actuel du système. Nous proposons un découpage de la preuve isolant cette difficulté.

(Abstract: pto)

*Samuel.Boutin@inria.fr

Unité de recherche INRIA Rocquencourt

Proving Correctness of the Translation from Mini-ML to the CAM with the Coq Proof Development System

Abstract: In this report we show how we proved correctness of the translation from a small applicative language with recursive definitions (Mini-ML) to the Categorical abstract machine (CAM) using the Coq system. Our aim was to mechanize the proof of J. Despeyroux [9]. Like her, we use natural semantics to axiomatise the semantics of our languages. We have only partially mecanised the proof. The semantics of the source and target languages use rational trees in their definition and at this stage the Coq system is inefficient in axiomatising such structures. We propose a presentation of the correctness proof to isolate this difficulty.

1 Introduction

Dans le domaine des preuves de programmes, la correction de la compilation des langages de programmation occupe une place privilégiée. En effet, le bon fonctionnement d'un exécutable dépend autant de la correction du programme dont il provient que de la correction du compilateur qui l'a produit.

Par ailleurs, comme les compilateurs sont toujours des programmes d'une taille imposante, leur correction est une tâche énorme qu'il importe de mécaniser.

L'idée est alors pour reprendre F. Pfenning dans [23] d'utiliser d'une part des systèmes d'aide à la démonstration (comme Coq par exemple) qui mécanisent la déduction dans des systèmes logiques et, d'autre part des spécifications sémantiques définies dans un style "système de déduction logique" (comme par exemple la sémantique naturelle [16]) pour mécaniser des preuves de correction du typage, de la compilation (...etc) des langages de programmation.

Dans [23],J. Hannan et F. Pfenning utilisent le langage Elf pour certifier la correction de la compilation du λ-calcul pur dans la CAM.

Nous proposons dans ce papier une preuve de correction de compilation d'un petit langage applicatif "Mini-ML" (représentant le noyau des langages fonctionnels) en code CAM (Categorical Abstract Machine) realisé dans le système d'aide à la démonstration Coq. Il s'agit pour nous de mécaniser complètement la preuve exposée dans [9]. Nous y parvenons modulo quelques petites modifications sur les sémantiques des langages source et cible que nous explicitons dans la section 2.

Cette preuve est réalisée dans la version 5.10 du système d'aide à la démonstration Coq. Nous supposerons que le lecteur est familiarisé avec le système Coq ([4], [5], [12], [15], [25]) et en conséquence nous focalisons notre rapport sur les problèmes de l'axiomatisation de la sémantique naturelle en Coq et sur la preuve de correction.

Après avoir rappelé brièvement d'une part quelques définitions sur la description formelle de la compilation, et d'autre part l'origine de la sémantique naturelle, nous présentons dans la deuxième section les langages sources et cibles ainsi que leurs sémantiques tels qu'ils sont présentés dans [3]. Nous précisons la partie de la preuve de correction que nous avons mécanisée avec Coq.

Nous décrivons dans la section 3 notre axiomatisation de la syntaxe et de la sémantique de Mini-ML. Dans la section 4, nous faisons de même pour la machine abstraite vers laquelle nous compilons Mini-ML. La section 5 décrit l'axiomatisation de la fonction de compilation et la preuve. La section 6 traite des tentatives similaires de mécanisation réalisées avec d'autres méthodes ou d'autres systèmes. La section 7 évalue les difficultés qu'offrent la réalisation d'une telle preuve et commente les avantages de la dernière version de Coq par rapport aux versions antérieures.

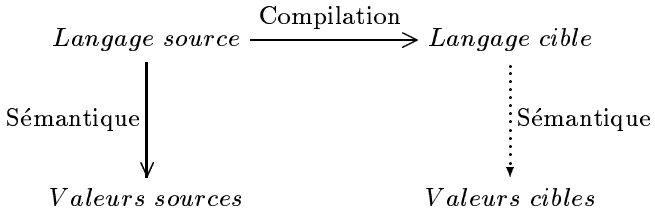
1.1 Description formelle de la compilation

Etant donnés deux langages de programmation, un langage source et un langage cible, la compilation peut être vue formellement comme une fonction qui prend en argument une phrase bien formée du langage source et rend une phrase bien formée du langage cible. On parle alors de compilation du langage source vers le langage cible.

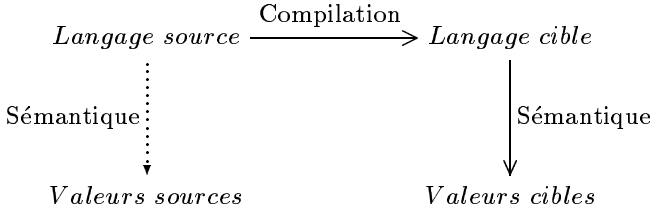
Si l'on veut modéliser la compilation, les langages sources et cibles doivent être chacun munis d'une sémantique qui modélise leur évaluation. On doit donc disposer d'un ensemble de valeurs sources et de valeurs cibles. On dira qu'un programme s'évalue s'il s'évalue d'après sa sémantique en une valeur.

Nous décrivons maintenant, en utilisant les diagrammes de L. Morris, trois critères essentiels de la compilation par rapport auxquels nous situerons notre travail. Dans [9], il est bien précisé que la formulation de ces propriétés dépend de la totalité, du déterminisme des fonctions sémantiques. Nous prenons volontairement des énoncés simplifiés.

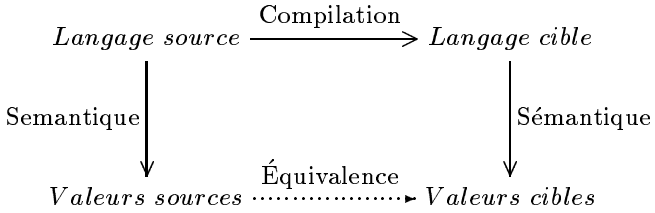
(a). **La complétude** (en anglais "completeness") désigne le fait que si un programme du langage source s'évalue, il en est de même de sa compilation dans le langage cible. Ceci correspond au dessin suivant où les traits en pointillés correspondent au but à prouver et les traits pleins aux hypothèses.



(b). **La cohérence** (en anglais "soundness") désigne le fait que si un programme P du langage source a un programme compilé qui s'évalue, alors P s'évalue.



(c). **La correction** (en anglais "correctness") désigne le fait que si un programme du langage source et son compilé s'évaluent tous les deux, alors ils s'évaluent en la même valeur modulo une équivalence entre valeurs sources et valeurs cibles à préciser.



Nous parlerons aussi de traduction pour désigner la fonction qui à une phrase bien formée du langage source associe une phrase bien formée du langage cible, c'est à dire la fonction de compilation.

1.2 Les sémantiques naturelles

En 1981, G. Plotkin a introduit un style de représentation des sémantiques opérationnel appelé *Structural Operational Semantics* dans [24] que nous abrègerons en SOS. Une SOS d'un langage est un système formel permettant de déduire des propositions de la forme : $\rho \vdash M \rightarrow M'$. M désigne alors un terme de notre langage, ρ désigne alors l'environnement dans lequel on précise la "signification" des variables libres du terme M, et M' est "la partie du terme M" qu'il reste à évaluer. Une telle proposition se lit dans "l'environnement ρ, le terme M s'évalue en M'".

Comme G. Plotkin le dit dans [24], une sémantique définie dans un style SOS peut être considérée comme un système de déduction naturelle "à la Gentzen" écrit dans un style **linéaire**.

Par "linéaire", il entend ici que les sémantiques SOS sont décrites dans un style "small step" et donc que la **preuve** d'un jugement (regardé logiquement comme un **séquent**) $\rho \vdash M \rightarrow M'$ sera une **séquence** de règles de sémantique, qui dénotent chacune une étape ("step") élémentaire ("small") de l'évaluation décrite.

L'introduction de ce formalisme a été très utile, bien au delà de la description des sémantiques opérationnelles. Il a été étudié en particulier par G. Kahn et ses collaborateurs dans [16] et [9]. Ces derniers ont proposé une forme de sémantique opérationnelle appelée sémantique naturelle (SN) où l'analogie avec les systèmes de déduction naturelle à la Gentzen est plus franche. La restriction "linéaire" de Plotkin disparaît et l'on parle de sémantique "big step" car dans une règle d'inférence peuvent maintenant se cacher plusieurs étapes d'évaluation mais aussi plusieurs stratégies d'évaluation. Les jugements $\rho \vdash M : V$ de SN sont

considérés comme des séquents, à ceci près que ρ n'est pas nécéssairement un **ensemble** d'"hypothèses" (Ceci est rappelé brièvement en section 5) ; ρ peut être une structure rationnelle représentant un arbre infini.

Pour conclure cette petite comparaison, nous présentons la règle de l'application de la sémantique opérationnelle de la CAM dans les deux styles. L'avantage de ce choix est que dans les deux cas les notations sont très proches : s et $s1$ désignent un tas de la CAM, C et C' des successions de commandes de la CAM, @ est l'opérateur de concaténation des commandes et ; l'opérateur de séquencement. Dans un style SOS cela donne :

$$\overline{((cur(C),\rho),V).s \quad app;C' \quad \rightarrow \quad (\rho,V).s \quad C@C'}$$

Dans un style SN on pourrait écrire :

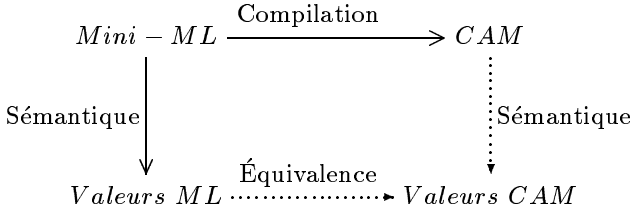
$$\frac{(\rho,v).s \quad \rightarrow_C \quad s_1}{((cur(c),\rho),v).s \quad \rightarrow_{app} \quad s_1}$$

Dans le cas d'une présentation SOS, on passe d'un état comprenant un tas et une suite de commandes à exécuter à l'état suivant où la première commande du tas a effectivement été exécutée ; cette présentation est "linéaire". Dans la présentation SN, un état correspond à un tas et l'on exécute une suite de commandes (en indice de la flèche). On voit que de l'état $((cur(c),\rho),v).s$ à l'état s_1, il peut y avoir eu de nombreux états intermédiaires (suivant la longueur de C) et que l'exécution de *app* peut elle-même faire partie d'une séquence de commandes ; cette présentation est "récursive".

C'est dans le style des SN que nous présentons la sémantique de Mini-ML et de notre machine abstraite à l'instar de J. Despeyroux dans [9] ou G. Kahn dans [16].

1.3 La preuve

Nous avons utilisé le système d'aide à la démonstration Coq pour démontrer la correction et la complétude de la compilation de Mini-ML en code CAM. Les sémantiques choisies pour notre axiomatisation sont toutes des sémantiques naturelles. Notre démonstration se résume au diagramme suivant :



Il est à remarquer que nous compilons des expressions non typées. C'est à dire que nous ne prenons pas en compte la phase initiale de vérification de types, aussi appelée **sémantique statique**. Une description précise des sémantiques est donnée en section 2.

2 Présentation Typol des langages sources et cibles

Avant de donner une description de Mini-ML et de la CAM dans Gallina, le langage de spécification de Coq, nous présentons sommairement les syntaxes et sémantiques des langages sources et cibles dans un style "Typol". Il existe plusieurs versions légèrement différentes de ces sémantiques et nous nous referrons précisément à [3]. Nous nous restreignons cependant à décrire le langage Mini-ML avec de simples variables et non pas des *patterns* comme cela est fait dans [3].

Sorts : EXP,Ident
constructors

$ident$:		$\rightarrow$	Ident
$true$:		$\rightarrow$	EXP
$false$:		$\rightarrow$	EXP
$number$:		$\rightarrow$	EXP
$apply$:	EXP $\times$ EXP	$\rightarrow$	EXP
$mlpair$:	EXP $\times$ EXP	$\rightarrow$	EXP
$lambda$:	Ident $\times$ EXP	$\rightarrow$	EXP
let :	Ident $\times$ EXP $\times$ EXP	$\rightarrow$	EXP
$letrec$:	Ident $\times$ Ident $\times$ EXP	$\rightarrow$	EXP
if :	EXP $\times$ EXP $\times$ EXP	$\rightarrow$	EXP

Figure 1: La syntaxe abstraite de Mini-ML

$$I \longleftarrow a.\rho \vdash^{\text{val\_of}} ident(I) : a \qquad (1)$$

$$\frac{\rho \vdash^{\text{val\_of}} ident(I) : a \quad , \quad K \neq I}{K \longleftarrow b.\rho \vdash^{\text{val\_of}} ident(I) : a} \qquad (2)$$

Figure 2: Evaluation des identificateurs : val_of

2.1 Présentation de Mini-ML :

La syntaxe abstraite de Mini-ML est présentée Figure 1.

La présentation de la sémantique de Mini-ML nécessite l'introduction d'une nouvelle catégorie syntaxique : celle des valeurs et environnements de Mini-ML qui sont définis mutuellement récursivement. Une valeur interprète un terme bien formé de Mini-ML dans un environnement donnant l'interprétation des variables libres de ce terme et nous présentons informellement la catégorie des valeurs dans cette optique :

• une valeur ML est donc soit un booléen ($true$ et $false$) ou un entier
-soit une paire de valeurs (α,β)
-soit une fermeture de la forme "$[\![\lambda P.E; \rho]\!]$" où "P" est un identificateur, "E" une expression ML et "ρ" un environnement ML.
-soit une fermeture opaque correspondant à un opérateur prédéfini.
• un environnement est soit vide ($\emptyset$)
-soit composé d'un couple variable-valeur et d'un environnement ("P $\leftarrow$ v.ρ" où "P" est un identificateur, "v" une variable et "ρ" un environnement)

La sémantique naturelle de Mini-ML est alors décrite en deux étapes : On décrit d'abord l'accès aux variables d'un environnement Mini-ML (Figure 2). Puis la sémantique des expressions Mini-ML à proprement parler (Figure 3).

Un environnement est une liste d'association dont les clés sont les identificateurs.

La sémantique naturelle de mini-ML est définie par récurrence sur le terme de Mini-ML qu'on évalue. La présentation "big-step" se présente sous forme de règles d'inférence dont les séquents "$\rho \vdash mlexp : mlval$" se lisent "l'expression ML, $mlexp$, s'évalue en la valeur ML, $mlval$, dans l'environnement ρ".

Dans la description suivante, ρ désigne un environnement ML ; α et β désignent des valeurs ML ; P et I désignent des identificateurs ; E, E_1 et E_2 désignent des expressions ML.

$$\rho \vdash \text{true} : true \tag{1}$$

$$\rho \vdash \text{false} : false \tag{2}$$

$$\rho \vdash \text{number(n)} : n \tag{3}$$

$$\rho \vdash \lambda P.E : [\![\lambda P.E; \rho]\!] \tag{4}$$

$$\frac{\rho \vdash^{\text{val\_of}} ident(I) \rightarrow \alpha}{\rho \vdash ident(I) : \alpha} \tag{5}$$

$$\frac{\rho \vdash E_1 : true \quad , \quad \rho \vdash E_2 : \alpha}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \alpha} \tag{6}$$

$$\frac{\rho \vdash E_2 : false \quad , \quad \rho \vdash E_3 : \alpha}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \alpha} \tag{7}$$

$$\frac{\rho \vdash E_1 : \alpha \quad , \quad \rho \vdash E_2 : \beta}{\rho \vdash (E_1, E_2) : (\alpha, \beta)} \tag{8}$$

$$\frac{\rho \vdash E_1 : \text{opaque(op)} \quad , \quad \rho \vdash E_2 : \alpha \quad , \quad \vdash^{eval} op, \alpha \ : \ \beta}{\rho \vdash E_1 \ E_2 \ : \ \beta} \tag{9}$$

$$\frac{\rho \vdash E_1 : [\![\lambda P.E, \rho_1]\!] \quad , \quad \rho \vdash E_2 : \alpha \quad , \quad P \longleftarrow \alpha.\rho_1 \vdash E : \beta}{\rho \vdash E_1 E_2 : \beta} \tag{10}$$

$$\frac{\rho \vdash E_2 : \alpha \quad , \quad P \longleftarrow \alpha.\rho \vdash E_1 : \beta}{\rho \vdash \text{let P= } E_2 \text{ in } E_1 : \beta} \tag{11}$$

$$\frac{\rho_1 \ = \ P \ \leftarrow \ [\![\lambda P.E; \rho_1]\!].\rho \quad , \quad \rho 1 \vdash E_2 : \alpha}{\rho \vdash \text{letrec } P = \lambda x.E \text{ in } E_2 : \alpha} \tag{12}$$

Figure 3: Sémantique de Mini-ML

$sorts$: $Value$, $Command$, $Commands$, $Program$

$subsorts$: $Command \subset Commands$

$Programs$:

$$\begin{array}{lll} program: & Commands & \rightarrow Program \\ coms: & Command^{\star} & \rightarrow Commands \end{array}$$

$Commands$:

$$\begin{array}{lll} quote: & Value & \rightarrow Command \\ op: & & Command \\ car: & & Command \\ cdr: & & Command \\ cons: & & Command \\ push: & & Command \\ swap: & & Command \\ app: & & Command \\ rplac: & & Command \\ cur: & Command & \rightarrow Command \\ branch: & Command \times Command & \rightarrow Command \end{array}$$

$Value$:

$$\begin{array}{ll} int: & Value \\ bool: & Value \\ null-value: & Value \end{array}$$

Figure 4: Langage de la CAM (Instructions et Valeurs)

2.2 Présentation de la Machine Catégorique Abstraite

La machine catégorique abstraite est une machine à pile dans la lignée de la machine SECD de Landin. Elle est cependant beaucoup plus simple que SECD car elle ne manipule qu'une pile (contre 4 pour son ancêtre). Elle est de plus théoriquement supportée par le paradigme catégorie-langage machine exposé dans l'article d'origine [6]. Nous présentons sa spécification syntaxique telle que décrite dans [3] en Figure 4.

La CAM est une machine à piles, les piles étant des listes de valeurs. Contrairement à Mini-ML, on ne peut pas distinguer de valeurs et d'environnements dans la CAM. C'est seulement en pensant à la compilation de Mini-ML en code CAM que l'on peut spéculer sur la nature d'une valeur CAM qui peut aussi bien représenter une valeur ML qu'un environnement ML. Cependant, même sans avoir introduit la traduction de Mini-ML en code CAM, et ne connaissant pas d'autre signification de la CAM que cette traduction, nous parlerons de valeurs et d'environnement CAM sachant que ces deux ensembles appartiennent à la même catégorie syntaxique. Un environnement CAM est simplement une liste de valeurs CAM construite à l'aide du constructeur de paires (De manière imagée, on peut dire sous forme de peigne : Par exemple l'environnement "$[\alpha, \beta, \gamma]$" est représentée par le couple $(((\emptyset, \alpha), \beta), \gamma))$ Dans la CAM, aucun identificateur n'apparaît puisque ces derniers sont transformés en indices de de Bruijn. Une "valeur" est :

-soit $true$ ou $false$, ou un entier ou un opérateur opaque.

-soit une fermeture $[\![C; \rho]\!]$, où "C" est une suite d'instructions et "ρ" un environnement CAM.

-soit une paire de valeurs CAM.

La sémantique de la CAM est décrite comme une suite de règles de transition d'un état vers un autre en Figure 5. Dans ces règles, "$s \rightarrow_C s_1$" doit se lire : "la commande C transforme la pile s en la pile s_1".

2.3 La traduction de Mini-ML en code CAM et l'équivalence entre valeurs Mini-ML et CAM

Il nous reste à introduire la traduction de Mini-ML vers la CAM (Figure 6). Dans ces règles de traduction, s désigne un squelette d'environnement c'est à dire la première projection d'une liste d'association ou encore la liste des identificateurs sous-jacente à un environnement. En fait, ce squelette est implicitement celui de l'environnement dans lequel s'évalue l'expression ML que l'on traduit. Nous ne précisons pas la fonction *Access* qui traduit une variable et un environnement en une suite de *car* suivie d'un *cdr* et dénotant l'emplacement de la variable dans (l'environnement ou) le squelette. C'est cette fonction qui donne une traduction "à la de Bruijn" de chaque identificateur. Pour les autres expressions de Mini-ML, la traduction est donnée sous forme d'un ensemble de règles d'inférence comme suit : C, C_1, C_2 et C_3 désignent des codes CAM. E, E_1, E_2 et E_3 désignent des expressions ML et, p et x, des variables.

On peut définir une relation d'équivalence entre valeurs de Mini-ML et valeurs CAM. Comme valeurs et environnements sont définis de manière mutuelle récursive, il est nécessaire de définir simultanément cette équivalence sur les valeurs et les environnements. En voici une présentation informelle :

- Les valeurs des booléens dans mini-ML et dans la CAM doivent se correspondre.
- Une paire de valeurs ML (α, β) est équivalente à une paire de valeurs CAM `(a,b)` si et seulement si α est équivalente à `a` et β est équivalente à `b`.
-des environnements ML et CAM se correspondent si les valeurs de l'environnement CAM et de l'environnement ML sont équivalentes et dans le même ordre (modulo la petite transformation entre paires et listes déjà mentionnée)
-Une fermeture ML et une fermeture CAM sont équivalentes si leurs codes respectifs se correspondent par la traduction et si leurs environnements respectifs sont équivalents.
-Un environnement bouclé de Mini-ML "$(\rho_1 = P \leftarrow [\![\lambda P.E; \rho_1]\!].\rho)$" est équivalent à l'environnement bouclé de la CAM $(\phi, (\phi_1 = [\![C; (\phi, \phi_1)]\!])$ où ϕ est équivalent à ρ et C est la traduction de E dans le squelette $x.P.\rho$. On remarquera au passage que si dans Mini-ML on "boucle dans les environnements", dans la CAM on "boucle dans les valeurs".

Dans la suite de cette section, on notera par exemple ML(10) la dixième règle de la sémantique de Mini-ML. On adoptera une notation similaire pour l'accès aux variables (val_of), la sémantique de la CAM (CAM) et les règles de la compilation (Trad).

2.4 Problèmes de représentation

Les sémantiques de Mini-ML et de la CAM peuvent être considérés comme des programmes Prolog (Puisque les sémantiques exposées sont du premier ordre ; on pourra s'en assurer en inspectant l'axiomatisation Coq dans les sections suivantes). C'était d'ailleurs utile dans le système Centaur puisqu'une telle traduction existait en Mu-Prolog au moment de la parution de [3]. Cependant, comme cela est remarqué dans [3], les sémantiques de Mini-ML et de la CAM ci dessus, inspirées par [6] utilisent des graphes pour interpréter la récursivité. Dans les deux règles suivantes extraites des sémantiques de Mini-ML et de la CAM, les environnements formés pour évaluer une expression récursive sont des arbres rationnels :

Dans les règles qui suivent, α et β désignent des valeurs CAM, ρ désigne un environnement de la CAM, $[\![C;\rho]\!]$ désigne une fermeture CAM, $[\![C;\rho]\!]_{rec}$ désigne une fermeture récursive,$COMS$ C, C_1 et C_2 désignent des suites d'instructions CAM. COM désigne une et une seule instruction CAM. Le constructeur "$coms$" (syntaxe abstraite du séquencement des instructions) se note ";" en notation infixe.

$$\frac{\text{init\_stack} \rightarrow_{COMS} \alpha.s}{\vdash Program(COMS) : \alpha} \quad (1)$$

$$s \rightarrow_{\emptyset} s \quad (2)$$

$$\frac{s \rightarrow_{COM} s_1, s_1 \rightarrow_{COMS} s_2}{s \rightarrow_{COM;COMS} s_2} \quad (3)$$

$$\alpha.s \rightarrow_{quote(bool(b))} b.s \quad (4)$$

$$\alpha.s \rightarrow_{quote(int(n))} n.s \quad (5)$$

$$\alpha.s \rightarrow_{quote(x)} x.s \quad (var(x)) \quad (6)$$

$$(\alpha,\beta).s \rightarrow_{car} \alpha.s \quad (7)$$

$$(\alpha,\beta).s \rightarrow_{cdr} \beta.s \quad (8)$$

$$\beta.\alpha.s \rightarrow_{cons} (\alpha,\beta).s \quad (9)$$

$$\alpha.s \rightarrow_{push} \alpha.\alpha.s \quad (10)$$

$$\alpha.\beta.s \rightarrow_{swap} \beta.\alpha.s \quad (11)$$

$$\frac{\vdash^{eval} OP, \alpha : \beta}{\alpha.s \rightarrow_{op(OP)} \beta.s} \quad (12)$$

$$\frac{s \rightarrow_{C_1} s_1}{true.s \rightarrow_{branch(C_1,C_2)} s_1} \quad (13)$$

$$\frac{s \rightarrow_{C_2} s_2}{false.s \rightarrow_{branch(C_1,C_2)} s_2} \quad (14)$$

$$\alpha.s \rightarrow_{cur(C)} [\![C;\rho]\!].s \quad (15)$$

$$\frac{(\alpha,\rho).s \rightarrow_{C} s_1}{([\![C;\rho]\!],\alpha).s \rightarrow_{app} s_1} \quad (16)$$

$$\frac{V = \rho_1}{(\rho,V).\rho_1.s \rightarrow_{rplac} (\rho,\rho_1).s_1} \quad (17)$$

Figure 5: Sémantique dynamique de la CAM : CAM

$$\frac{init\_pat \vdash E \rightarrow C}{\vdash E : program(C)} \quad (1)$$

$$true \rightarrow_s quote(bool(true)) \quad (2)$$

$$false \rightarrow_s quote(bool(false)) \quad (3)$$

$$number(n) \rightarrow_s quote(int(n)) \quad (4)$$

$$\frac{(Access\ p\ s\ C)}{ident(I) \rightarrow_s C} \quad (5)$$

$$\frac{E_1 \rightarrow_s C_1, E_2 \rightarrow_s C_2, E_3 \rightarrow_s C_3}{\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \rightarrow_s push; C_1; branch(C_2, C_3)} \quad (6)$$

$$\frac{E_1 \rightarrow_s C_1, E_2 \rightarrow_s C_2}{(E_1, E_2) \rightarrow_s push; C_1; swap; C_2; cons} \quad (7)$$

$$\frac{E_1 \rightarrow_s C_1, E_2 \rightarrow_s C_2}{E_1 E_2 \rightarrow_s push; C_1; swap; C_2; cons; app} \quad (8)$$

$$\frac{E_2 \rightarrow_s C_2, \vdash^{trans\_const} E_1 : C_1}{E_1 E_2 \rightarrow_s C_2; C_1} \quad (9)$$

$$\frac{E_1 \rightarrow_s C_1, E_2 \rightarrow_{p.s} C_2}{\text{let } p{=}E_1 \text{ in } E_2 \rightarrow_s push; C_1; cons; C_2.} \quad (10)$$

$$\frac{E \rightarrow_{x.p.s} C, E_2 \rightarrow_{p.s} C_2}{\text{letrec } p{=}\lambda x.E \text{ in } E_2 \rightarrow_s push; quote(\rho 1); cons; push; cur(C); swap; rplac; C_2.} \quad (11)$$

$$\frac{E \rightarrow_{p.s} C}{\lambda p.E \rightarrow_s cur(C).} \quad (12)$$

Figure 6: Compilation de Mini-ML vers la CAM : “Trad”

$$\frac{\rho_1 \ = \ P \ \leftarrow \ [\![\lambda P.E; \rho_1]\!].\rho, \rho 1 \vdash E_2 : \alpha}{\rho \vdash \text{letrec } P = \lambda x.E \text{ in } E_2 : \alpha} \quad ML(12)$$

$$\frac{V = \rho_1}{(\rho, V).\rho_1.s \rightarrow_{rplac} (\rho, \rho_1).s_1} \quad CAM(17)$$

Cependant, la représentation de tels arbres pose des problèmes aussi bien dans Coq que dans Prolog. C'est seulement l'absence d'occur-check dans Mu-Prolog qui peut permettre d'utiliser un programme comme `ML_DS`, mais un tel usage n'est pas légitime.

Une alternative est alors proposée dans [3] pour définir une sémantique équivalente de Mini-ML sans structures bouclées. Elle correspond à créer des environnements dépliables. Cependant, toujours dans [3], Il n'est pas remarqué qu'en changeant la sémantique de Mini-ML, la fonction de traduction du code Mini-ML en code CAM reste valide, mais les structures bouclées ne sont pas éliminées de la CAM ! Aucune proposition n'est d'ailleurs faite pour éliminer les structures bouclées de la CAM dans [9], [3] ou [16]. Or, quand on désire éliminer les structures rationnelles de la CAM, il faut le faire en fonction des modifications apportées lors de cette même opération sur Mini-ML. En effet, intuitivement, si l'on considère la CAM comme une machine décomposant les étapes de l'évaluation d'expressions Mini-ML, il est normal qu'en changeant la sémantique de Mini-ML, on ait des restrictions en changeant celle de la CAM. Dans les deux sous sections suivantes, nous décrivons les sémantiques de Mini-ML et de la CAM lorsque l'on veut boucler dans les environnements ou dans les fermetures respectivement. Pour des raisons chronologiques, c'est cette dernière approche qui a été utilisée dans ce rapport, mais les deux approches sont également viables.

2.5 Dépliage dans les environnements

Dans [3], l'utilisation d'environnements dépliables (ou récursifs) est proposée comme alternative aux graphes.

Un environnement récursif sensé simuler la boucle du letrec est de la forme (rec p λx.E ρ) où p et x sont des identificateurs, E une expression ML et ρ un environnement. "rec" est alors un nouveau constructeur d'environnements. le dépliage se fait lors de l'accès aux variables dans l'environnement (ensemble de règles `val_of`). Par exemple, lors de l'accès à l'identificateur "fact", on remet dans la fermeture correspondante un environnement récursif. Ceci s'exprime par la règle suivante :

$$\frac{P \longleftarrow [\![\lambda x.E, (rec\ P\ \lambda x.E\ \rho)]\!].\rho \vdash^{\text{val\_of}} ident(I) : a}{(rec\ P\ \lambda x.E\ \rho) \vdash^{\text{val\_of}} ident(I) : a} \quad val\_of(3)$$

la sémantique de la récursion devient alors

$$\frac{(rec\ P\ \lambda x.E\ \rho) \vdash E_2 : \alpha}{\rho \vdash letrec\ P\ =\ \lambda x.E\ in\ E_2 : \alpha} \quad ML(12')$$

Le "coût" de l'élimination du graphe est donc alors l'ajout d'un nouveau constructeur d'environnement et d'une règle d'accès aux variables dans l'environnement ainsi qu'une modification de la sémantique du letrec. L'équivalence des sémantiques avec ou sans graphe est démontrée dans [3].

Comme nous l'avons dit précédemment, à toute modification de la sémantique de Mini-ML correspond une modification de la sémantique de la CAM. Ici, la sémantique avec graphe de la CAM est toujours pertinente vis à vis de la sémantique sans graphes de Mini-ML. Mais l'utilisation de graphes étant à proscrire, il nous faut éliminer les graphes de la sémantique de la CAM. La seule solution satisfaisante est alors de boucler aussi dans les environnements de la CAM. Pour cela, il faut rajouter un constructeur d'"environnements récursifs", appelons le *rec*, et étendre la sémantique des commandes accédant à l'environnement pour permettre le dépliage : ($Rec\ C\ \rho$) dénotera l'environnement récursif et l'accés à la de Bruijn dans un environnement récursif est donné par les règle supplémentaires :

$$(Rec\ C\ \rho).s \rightarrow_{car} \rho.s \qquad CAM(18)$$

$$(Rec\ C\ \rho).s \rightarrow_{cdr} [\![C;(Rec\ C\ \rho)]\!].s \quad CAM(19)$$

La règle du *rplac* ainsi que l'instruction *rplac* sont supprimées en retour. Néanmoins, il faut rajouter une instruction de création des environnements récursifs : Nous proposons par exemple la création d'une instruction $fold$ dont le comportement est décrit par la sémantique :

$$\rho.s \rightarrow_{fold(COMS)} (Rec\ COMS\ \rho).s \quad CAM(20)$$

La traduction d'une instruction récursive devient :

$$\frac{E \rightarrow_{x.p.s} C, E_2 \rightarrow_{p.s} C_2}{\text{letrec } p{=}\lambda x.E \text{ in } E_2 \rightarrow_s fold(cur(C)); C_2} \quad Trad(11')$$

Nous n'avons pas utilisé cette version de la CAM dans notre développement et ne demontrerons pas l'équivalence des machines (avec ou sans graphes) ici.

2.6 Dépliage dans les fermetures

Il existe une alternative au dépliage "dans les environnements", c'est le "dépliage dans les valeurs".

C'est cette méthode que nous avons utilisée pour faire notre preuve. Il va de soi qu'elle n'est pas meilleures que la précédente, c'est une alternative qui a été choisie pour des raisons chronologiques de développement. Nous démontrons en appendice les équivalences entre les sémantiques avec graphe et les sémantiques sans graphes de Mini-ML et de la CAM .

Relativement au langage Mini-ML, déplier dans les fermetures ou "boucler dans les valeurs", au contraire de "boucler dans les environnements", signifie qu'on ajoute des fermetures récursives à la catégorie des valeurs Mini-ML (alors que dans le paragraphe précédent on avait ajouté des environnements récursifs).

Et au lieu de déplier l'environnement lors de l'évaluation d'un identificateur, on va maintenant "déplier" une fermeture récursive lors de l'"application".

Concrètement, cela signifie qu'on a une nouvelle sorte de valeur ML, les fermetures récursives de la forme $[\![P;\lambda x.E;\rho]\!]_{rec}$. Dans la sémantique de Mini-ML, il faut supprimer l'ancienne règle de la récursion et la remplacer par :

$$\frac{P \leftarrow [\![P\ ;\lambda x.E;\rho]\!]_{rec}.\rho \vdash E_2 : \alpha}{\rho \vdash letrec\ P\ =\ \lambda x.E\ in\ E_2 : \alpha} \quad ML(12'')$$

Il faut aussi ajouter une règle d'application éliminant les fermetures récursives :

$$\frac{\rho \vdash E_1 : [\![P\ \lambda x.E;\rho_1]\!]_{rec}, \rho \vdash E_2 : \alpha, x \leftarrow \alpha, P \leftarrow [\![P\ \lambda x.E;\rho_1]\!]_{rec}.\rho_1 \vdash E : \beta}{\rho \vdash E_1\ E_2 : \beta} \quad ML(10bis)$$

Il faut maintenant procéder à une transformation similaire de la CAM.

On définit donc une nouvelle espèce de valeurs CAM, les fermetures récursives de la CAM, que l'on note $[\![C;\rho]\!]_{rec}$ et une instruction cur_{rec} qui introduit de telles fermetures dans l'environnement avec la règle :

$$\rho.s \rightarrow_{cur_{rec}(COM)} [\![COM;\rho]\!]_{rec}.s \quad CAM(18)$$

L'instruction rplac disparait alors ainsi que sa règle associée dans la sémantique de la CAM. Mais il faut encore ajouter une règle d'application supplémentaire (celle qui effectue le dépliage).

$$\frac{((\rho, [\![C;\rho]\!]_{rec})\alpha).s \rightarrow_C s_1}{(\rho, [\![C;\rho]\!]_{rec}).s \rightarrow_{app} s_1} \quad CAM(16bis)$$

La fonction de traduction doit elle aussi être modifiée en conséquence, c'est à dire qu'il faut supprimer l'ancienne traduction du letrec et la remplacer par une nouvelle :

$$\frac{E \rightarrow_{x.p.s} C, E_2 \rightarrow_{p.s} C_2}{\text{letrec } p{=}\lambda x.E \text{ in } E_2 \rightarrow_s push; cur_{rec}(C); cons; C_2} \quad Trad(11'')$$

En conclusion de cette section, on peut dire que dorénavant, après le travail préliminaire de suppression des boucles la traduction en Coq d'un ensemble de prédicats du premier ordre ne pose aucun problème. La preuve de correction est elle aussi assez simple.

2.7 Nouveau profil de la preuve de correction

À l'aide des remarques de la section précédente, on peut donc découper en plusieurs étapes notre preuve de correction.

Soit ML' la sémantique de Mini-ML avec graphes. Soit ML la sémantique de Mini-ML sans graphes mais avec fermetures récursives. Soient $V_{ML'}$ et V_{ML} leurs ensembles de valeurs respectifs. Soient CAM' et (CAM') la CAM originelle (avec graphes) et sa sémantique ; Soit $V_{CAM'}$ l'ensemble des valeurs et environnements associés. Soient CAM et (CAM) la CAM avec cur_{rec} et sa sémantique ; et V_{CAM} l'ensemble des valeurs et environnements associés. Soient enfin T_{CAM} la traduction de CAM vers CAM'

Notre preuve de correction se décompose suivant le diagramme :

$$\begin{array}{ccccccc}
Mini-ML & \xrightarrow{ID} & Mini-ML & \xrightarrow{T} & CAM & \xrightarrow{T_{CAM}} & CAM' \\
\Big\downarrow{\scriptstyle (ML')} & & \Big\downarrow{\scriptstyle (ML)} & & \Big\downarrow{\scriptstyle (CAM)} & & \Big\downarrow{\scriptstyle (CAM')} \\
V_{ML'} & \xrightarrow{t_{ml}} & V_{ML} & \xrightarrow{t} & V_{CAM} & \xrightarrow{t_{cam}} & V_{CAM'}
\end{array}$$

Seul le carré central est pour l'instant prouvé automatiquement dans Coq et fait l'objet des sections suivantes. Nous présentons en appendice les preuves "manuelles" des carrés externes.

3 Axiomatisation de Mini-ML dans le système Coq

Nous présentons dans cette section l'axiomatisation du langage Mini-ML présenté en section 2. Nous évoquons les petites différences qui peuvent apparaitre par rapport à [3], [9] ou [16].

3.1 Axiomatisation de la syntaxe de Mini-ML

Les variables de Mini-ML sont nommées, nous choisissons `nat`, l'ensemble des entiers naturels défini dans le prélude standard du système Coq comme ensemble des variables.

```
Definition Pat = nat.
```

`nat` vérifie les deux propriétés qu'on peut attendre d'un ensemble de variables : il est infini et surtout l'égalité y est décidable.

On choisit arbitrairement un ensemble de constantes pour Mini-ML : les entiers et les booléens qui sont définis dans le Prélude standard du système. On se donne un ensemble d'opérateurs binaires sur les entiers à valeurs dans les entiers, qu'on indice par des éléments d'un ensemble `OP` :

```
Parameter OP:Set.
```

On se donne aussi une sémantique pour ces opérateurs :

```
Parameter eval_op:OP->nat->nat->nat.
```

Ces opérateurs servent de témoins ; leur arité et leur type sont choisis avec un souci de simplicité. Nulle part dans la preuve, on ne se sert de propriétés sur les entiers vus comme constantes du langages. On pourrait donc choisir un ensemble de constantes quelconque à leur place. De même l'arité des opérateurs est arbitraire.

Le concept de "type inductif" dans Coq nous permet alors de définir la syntaxe de Mini-ML comme une algèbre libre multisortée dont les constructeurs modélisent les constructions syntaxiques du langage :

```
Inductive Set MLexp=

  Bool:bool->MLexp
 |Num:nat->MLexp
 |op:OP->MLexp
 |id:Pat->MLexp
 |appl:MLexp->MLexp->MLexp
 |mlpair:MLexp->MLexp->MLexp
 |lambda:Pat->MLexp->MLexp
 |let':Pat->MLexp->MLexp->MLexp
 |letrec:Pat->Pat->MLexp->MLexp->MLexp
 |ite:MLexp->MLexp->MLexp->MLexp.
```

Par exemple, `(ite (Bool true) (lambda (id (S O)) (id (S O))) (lambda (id O) (id O)))` modélise l'expression ML "$if\ true\ then\ \lambda y.y\ else\ \lambda x.x$".

3.1.1 Remarques

1. Les entiers naturels sont surchargés dans cette axiomatisation ; notons que `(Num (S O))` représente l'entier `(S O)`, `(id (S O))` représente une variable, `(id (S (S O))` est une autre variable distincte de la précédente. Et même si nos variables sont identifiées à des entiers, nous n'utilisons pas les indices de de Bruijn.

2. Nous ne définissons pas de réductions pour le langage Mini-ML, on n'a donc pas besoin de définir de renommage. Comme l'étude de la sémantique le montre, la capture des variables s'éffectue par trois opérateurs liants :

   • Dans l'expression `(lambda P E)`, `P` est lié par l'opérateur `lambda` dans `E`.

   • Dans l'expression `(let' P E1 E2)`, `P` est lié par l'opérateur `let'` dans `E2`.

   • Dans l'expression `(letrec P x E1 E2)`, `x` est lié par l'opérateur `letrec` dans `E1` et `P` est lié par l'opérateur `letrec` dans `E2`.

3. Dans la syntaxe, la règle de formation du "letrec" diffère de [9]. En effet dans ce rapport, le "letrec" ne permet de définir récursivement que des fonctions, mais c'est dans la déclaration de la sémantique et des valeurs que l'on s'en rend compte : en analysant la sémantique d'une expression de la forme $letrec\ P = E1\ in\ E2$ (règle 12 de [9]), $E1$ devient le membre droit d'une fermeture ce qui exige d'après la règle de formation des fermetures qu'il soit de la forme $\lambda P.E$. Nous préferons déclarer effectivement `letrec P x E1 E2` qui sous-entend que `P` est lié à la fonction "$\lambda x.E1$" dans l'expression `E2`. De cette manière, même si l'on ne s'intéresse pas au typage ici, on ne permet pas de former des éléments récursifs bien typés qui n'ont aucun sens à la compilation (comme une liste infinie de "1" par exemple). Ce parti n'est pas celui de [16] qui est plus libéral et permet de définir n'importe quel objet récursif. Par exemple, $letrec\ P\ =\ (1, P)\ in\ P$ est accepté par les règles de la sémantique de Mini-ML. (Cette expression serait de plus bien typée si une structure de liste était déclarée, on remplacerait $(1, P)$ par $1 :: P$).

3.2 Axiomatisation de la sémantique naturelle de Mini-ML

Comme nous l'avons précisé dans l'introduction, la sémantique de Mini-ML choisie pour faire la preuve est la sémantique naturelle présentée en section 2. On axiomatise d'abord les valeurs et environnements, puis les règles d'évaluation d'un identificateur et enfin la sémantique.

3.2.1 Valeurs et environnements de Mini-ML

Nous axiomatisons ici la définition de la section 2.1, proche des définitions de [16] ou [9]. On définit à l'aide d'une définition mutuellement récursive les valeurs et les environnements de Mini-ML dans Coq. En effet, une fermeture contient un environnement et la valeur associée à une variable dans un environnement peut être une fermeture :

```
Mutual Inductive
MLval : Set :=
   num: nat -> MLval
 | boolean : bool -> MLval
 | valpair: MLval -> MLval -> MLval
 | OP_clos: OP -> MLval
 | Clos: Pat -> MLexp -> MLenv -> MLval
 | Clos_rec: Pat -> MLexp -> Pat -> MLenv -> MLval
with
MLenv : Set :=
   Enil: MLenv
 | Econs: Pat -> MLval -> MLenv -> MLenv.
```

Les valeurs sont donc les valeurs des constantes (entiers, booléens), des paires de valeurs, des fermetures associées aux opérateurs prédéfinis sur les constantes, des fermetures ou des fermetures récursives. Relativement aux fermetures, elles sont de deux sortes et l'on est pas fidèle ici à la sémantique présentée dans [9] ou [16] (cf section 2.4).

Les environnements Mini-ML sont des listes d'associations liant des identificateurs à des valeurs Mini-ML.

3.2.2 La sémantique de Mini-ML

Nous axiomatisons la sémantique de Mini-ML par le prédicat `ML_DS` qui est défini dans Coq comme un prédicat inductif. Chaque constructeur du prédicat correspond alors à une règle d'inférence de la description Typol dans [9]. Le jugement "$Env \vdash Exp : V$" s'exprime par la formule de vernaculaire Coq : "`(ML_DS Env Exp V)`". Informellement, dans un environnement Env où l'on donne les valeurs des variables libres du terme Exp, on donne des règles pour calculer la valeur V de Exp.

1. **Évaluation d'un identificateur**

   Avant de préciser `ML_DS`, il est nécessaire de décrire comment on accède à la valeur d'un identificateur dans un environnement. Encore une fois, le système de règles d'inférence définissant le jugement "$Env \vdash Var : Val$" est traduit naturellement dans Coq en un prédicat inductif :

```
Inductive Definition VAL_OF:MLenv->Pat->MLval->Prop:=

 ELT:(e:MLenv)(I:Pat)(a:MLval)
            (VAL_OF (Econs I a e) I a)

 |CHG:(e:MLenv)(X,I:Pat)(a,b:MLval)
        (VAL_OF e I a)->
                    ~(X=I)->
            (VAL_OF (Econs X b e) I a).
```

Notons qu'on utilise ici dans le constructeur `CHG` le fait que l'égalité sur les variables est décidable. Dans [9] et [16], le jugement équivalent *val_of* permet d'évaluer aussi n'importe quel filtre fabriqué à l'aide de variables et de l'opérateur de paire : ceci permet de définir des fonctions mutuellement récursives. Nous n'avons pas cette possibilité avec notre axiomatisation. Par rapport à l'axiomatisation de [16], on remarque que `~(X=I)` est une condition et toutes les expressions de la forme `(VAL_OF ...)` des séquents dans la terminologie "sémantique naturelle". Pour chaque règle (`ELT`, `CHG`) la dernière instance de `VAL_OF` joue le rôle "de séquent du bas" dans les règles d'inférence de la sémantique naturelle.

2. **`VAL_OF` est déterministe**

   Le prédicat `VAL_OF` associe une valeur Mini-ML et une seule à un couple composé d'un environnement et d'une variable. Le lemme suivant en atteste :

```
Lemma determ_VAL_OF :  (e:MLenv)(i:Pat)(V,V':MLval)
                (VAL_OF e i V')->
                    (VAL_OF e i V)->
                          V=V'.
```

   `VAL_OF` définit la sémantique d'un identificateur. Lors de la preuve de correction, concernant la compilation d'un identificateur, on utilise effectivement ce déterminisme.

3. **Évaluation d'un terme de Mini-ML**

   `ML_DS` est défini comme un prédicat inductif, notre axiomatisation est proche de celle définie dans [9] ou [16].

```
Inductive Definition ML_DS :MLenv->MLexp->MLval->Prop:=

  BOOL:(b:bool)(e:MLenv)(ML_DS e (Bool b) (boolean b))

 |NUM:(n:nat)(e:MLenv)(ML_DS e (Num n) (num n))

 |Sem_OP:(c:OP)(e:MLenv)(ML_DS e (op c) (OP_clos c))

 |LAMBDA:(e:MLenv)(P:Pat)(E:MLexp)
              (ML_DS e (lambda P E) (Clos P E e))

 |IDENT:(e:MLenv)(v:MLval)(I:Pat)
              (VAL_OF e I v)->
                 (ML_DS e (id I) v)

 |ITE1:(e:MLenv)(E1,E2,E3:MLexp)(v:MLval)
              (ML_DS e E1 (boolean true))->
                 (ML_DS e E2 v)->
                    (ML_DS e (ite E1 E2 E3) v)

 |ITE2:(e:MLenv)(E1,E2,E3:MLexp)(v:MLval)
              (ML_DS e E1 (boolean false))->
                 (ML_DS e E3 v)->
                    (ML_DS e (ite E1 E2 E3) v)

 |MLPAIR:(e:MLenv)(E1,E2:MLexp)(u,v:MLval)
              (ML_DS e E1 u)->
                 (ML_DS e E2 v)->
                    (ML_DS e (mlpair E1 E2) (valpair u v))
```

```
|APPml1:(e,e1:MLenv)(P:Pat)(E,E1,E2:MLexp)(u,v:MLval)
              (ML_DS e E1 (Clos P E e1))->
                (ML_DS e E2 u)->
                  (ML_DS (Econs P u e1) E v)->
                     (ML_DS e (appl E1 E2) v)

|APPml2:(e,e1:MLenv)(x,P:Pat)(E,E1,E2:MLexp)(u,v:MLval)
              (ML_DS e E1 (Clos_rec x E P e1))->
                (ML_DS e E2 u)->
                  (ML_DS (Econs x u (Econs P (Clos_rec x E P e1) e1)) E v)->
                    (ML_DS e (appl E1 E2) v)

|APPml_op:(e:MLenv)(E1,E2:MLexp)(n,m:nat)(c:OP)
          (ML_DS e E1 (OP_clos c))->
              (ML_DS e E2 (valpair (num n) (num m)))->
                    (ML_DS e (appl E1 E2) (num (eval_op c n m)))

|Sem_let:(e:MLenv)(P:Pat)(E1,E2:MLexp)(u,v:MLval)
              (ML_DS e E2 u)->
                 (ML_DS (Econs P u e) E1 v)->
                    (ML_DS e (let' P E2 E1) v)

|Sem_letrec:(e:MLenv)(P:Pat)(x:Pat)(E,E2:MLexp)(u:MLval)
              (ML_DS (Econs P (Clos_rec x E P e) e) E2 u)->
                 (ML_DS e (letrec P x E E2) u).
```

En définissant `ML_DS` de cette façon, on peut considérer ce prédicat comme un programme Prolog défini par l'ensemble de règles (nous garderons cette terminologie) : `BOOL`, `NUM`, ... `Sem_letrec`. En particulier, toute les règles sont du premier ordre. Ceci est d'ailleurs vrai pour tous les prédicats inductifs définis pour notre preuve de correction. Cette analogie est par ailleurs exploitée quand on interface le langage Typol avec Mu-Prolog (puisqu'on traduit alors directement un programme Typol en un programme Prolog).

Dans Coq, un schéma d'élimination associé à `ML_DS` est engendré automatiquement. Il exprime que seules les règles ci-dessus permettent de construire des habitants du type `ML_DS`. On peut par exemple montrer que `ML_DS` implique son complété de Clark (en utilisant une terminologie Prolog) ou peut être inversé (suivant une terminologie d'utilisateur de Coq). en d'autres termes, la formule `(ML_DS e E V)` est nécessairement engendrée par une des règles précédentes (d'où la possibilité de raisonner par récurrence sur la dernière règle appliquée).

Dans notre définition de `ML_DS`, en plus du contexte dans lequel ce "prédicat" est déclaré, nous différons de [9] et [16] en plusieurs points. D'abord, les règles concernant la sémantique d'une définition récursive diffèrent (la règle `Sem_letrec` ici, (11) dans [16] et (12) dans [9]). Ensuite, suite à notre codage de la récursivité nous avons une règle d'application supplémentaire (`APPml2` correspond à ML(12")).

4. **Le prédicat `ML_DS` est déterministe**

On peut montrer dans Coq le lemme suivant :

```
Lemma ML_DS_determ :
  (e:MLenv)(E:MLexp)(V:MLval)(ML_DS e E V)->
              (V':MLval)
                 (ML_DS e E V')->
                         V=V'.
```

Autrement dit, dans un environnement Mini-ML donné, notre sémantique associe une valeur et une seule à une expression Mini-ML.

4 La Machine Abstraite Catégorique

Comme nous l'avons fait pour le langage "Mini-ML" dans la section précédente, nous présentons la syntaxe puis la sémantique de la machine abstraite vers laquelle nous compilons Mini-ML. Pour une introduction à la "Categorical Abstract Machine", on pourra consulter l'article d'origine [6]. La CAM est une machine abstraite dans la lignée de SECD, la machine à pile de Landin. Elle consiste en un ensemble de "commandes" qui manipulent un tas de "valeurs". La plupart des commandes agissent sur le premier élément du tas mais certaines vont plus loin comme la commande "swap" qui intervertie les deux premiers éléments du tas. Si l'on considère la CAM comme un langage de programmation, on associe les commandes de la CAM à la syntaxe du langage, les valeurs du tas à l'ensemble des valeurs sémantiques du langage et l'action des commandes sur le tas à la sémantique du langage. Ici, syntaxe, valeurs et sémantique dans notre axiomatisation font en fait partie intégrante de la définition de la machine abstraite.

4.1 Syntaxe de la CAM

4.1.1 Les constantes de la CAM

Nous définissons d'abord les constantes de la CAM à l'aide du type `Value`.

```
Inductive Set Value:=
 null:Value
|elem:bool->Value
|int:nat->Value
|def_op:OP->Value.
```

Cet ensemble de valeurs est parfaitement arbitraire, on a choisi de prendre les entiers, les booléens et les opérateurs constants pour pouvoir compiler les constantes déclarées dans Mini-ML et être fidèle aux spécifications présentées en (2.2).

4.1.2 Les commandes de la CAM

On reprend principalement les notations de [6].

```
Inductive Set Commande:=
  quote:Value->Commande
 |car:Commande
 |cdr:Commande
 |cons:Commande
 |push:Commande
 |swap:Commande
 |branch:Commande->Commande->Commande
 |cur:Commande->Commande
 |cur_rec:Commande->Commande
 |app:Commande
 |o:Commande->Commande->Commande.
```

Par rapport aux présentations habituelles, on a internalisé la composition des commandes (l'opérateur `o` qui correspond à la séquentialité des commandes noté usuellement " ;"). On a aussi changé le codage de la récursivité. La commande habituelle rplac est ici remplacée par la commande `cur_rec`. Ces modifications sont justifiées dans la section 2. En plus des commandes "catégoriques" correspondant au λ-calcul pur, on a donc les commandes `quote`, `branch` et `cur_rec` qui correspondent respectivement à l'introduction des constantes, d'une instruction conditionnelle et de la récursivité.

4.2 Sémantique de la CAM

4.2.1 Valeurs et environnements pour la CAM

Les valeurs de la CAM sont définies inductivement :

```
Inductive Set CSem_val:=
 val:Value->CSem_val
 |Cam_pair:CSem_val->CSem_val->CSem_val
 |Cam_clos:Commande->CSem_val->CSem_val
 |Cam_clos_rec:Commande->CSem_val->CSem_val
 |Cam_nil:CSem_val.
```

En fait, comme on le verra dans la section présentant la preuve de correction, cet ensemble de valeurs permet de traduire non seulement les valeurs, mais aussi les environnements de Mini-ML grâce à l'opérateur de paire (`Cam_pair`) et à la constante "vide" (`Cam_nil`).

4.2.2 La sémantique de la CAM

Avant de définir la sémantique des commandes, il faut définir les tas de valeurs. Ceux sont des listes ; notre terminologie est relative à la sémantique de la CAM.

```
Inductive Set Etat:=
   nil:Etat
  |ETcons:CSem_val->Etat->Etat.
```

On peut maintenant définir la sémantique de la CAM. La présentation à la G. Kahn est "big step" : à un "Etat" (un "tas" de la machine abstraite), et à une suite de commandes, on associe un nouvel "Etat". Nous la définissons dans Coq à l'aide d'un prédicat inductif. Notons que notre définition est du premier ordre, et donc assimilable à un programme Prolog.

```
Inductive Definition CAM_DS:Etat->Commande->Etat->Prop:=


  QUO:(s:Etat)(a:CSem_val)(b:Value)
        (CAM_DS (ETcons a s) (quote b) (ETcons (val b) s))

 |CAR:(s:Etat)(a,b:CSem_val)
        (CAM_DS (ETcons (Cam_pair a b) s) car (ETcons a s))

 |CDR:(s:Etat)(a,b:CSem_val)
        (CAM_DS (ETcons (Cam_pair a b) s) cdr (ETcons b s))

 |CONS:(s:Etat)(a,b:CSem_val)
        (CAM_DS (ETcons b (ETcons a s)) cons (ETcons (Cam_pair a b) s))

 |PUSH:(s:Etat)(a:CSem_val)
        (CAM_DS (ETcons a s) push (ETcons a (ETcons a s)))

 |SWAP:(s:Etat)(a,b:CSem_val)
        (CAM_DS (ETcons a (ETcons b s)) swap (ETcons b (ETcons a s)))

 |BRANCHT:(s,s1:Etat)(c1,c2:Commande)
        (CAM_DS s c1 s1)->
            (CAM_DS (ETcons (val (elem true)) s) (branch c1 c2) s1)
```

```
|BRANCHF:(s,s2:Etat)(c1,c2:Commande)
       (CAM_DS s c2 s2)->
           (CAM_DS (ETcons (val (elem false)) s) (branch c1 c2) s2)

|CUR:(s:Etat)(a:CSem_val)(c:Commande)
       (CAM_DS (ETcons a s) (cur c) (ETcons (Cam_clos c a) s))

|APPcam1:(s,s1:Etat)(a,b:CSem_val)(c:Commande)
       (CAM_DS (ETcons (Cam_pair a b) s) c s1)->
           (CAM_DS (ETcons (Cam_pair (Cam_clos c a) b) s) app s1)
|APPcam2:(s,s1:Etat)(a,b:CSem_val)(c:Commande)
   (CAM_DS (ETcons (Cam_pair (Cam_pair b (Cam_clos_rec c b)) a) s) c s1)->
           (CAM_DS (ETcons (Cam_pair (Cam_clos_rec c b) a) s) app s1)

|APPcam_op:(s:Etat)(n,m:nat)(oper:OP)
    (CAM_DS (ETcons (Cam_pair (val (def_op oper))
      (Cam_pair (val (int n)) (val (int m)))) s) app
          (ETcons (val (int (eval_op oper n m))) s))

|CUR_REC:(s:Etat)(a:CSem_val)(c:Commande)
     (CAM_DS (ETcons a s) (cur_rec c) (ETcons (Cam_clos_rec c a) s))

|o_DS:(s,s1,s2:Etat)(c1,c2:Commande)
      (CAM_DS s c1 s1)->
          (CAM_DS s1 c2 s2)->
              (CAM_DS s (o c1 c2) s2).
```

`(CAM s1 C s2)` se lit : la commande C agit sur l'état s1 et le transforme en l'état s2. Excepté pour la récursion (règles `APPcam2` et `CUR_REC`) , nous sommes fidèles aux présentations usuelles.

5 La preuve de correction

5.1 La fonction de compilation

Pour présenter la compilation de Mini-ML en code CAM on procède en deux temps. On définit d'abord la traduction d'un identificateur de Mini-ML en code CAM. Puis les autres constructions syntaxiques de Mini-ML sont traduites en suites de commandes CAM. Il faut donc pour axiomatiser la traduction, commencer par définir la correspondance entre environnements de Mini-ML et environnements de la CAM et la traduction d'un identificateur de Mini-ML en un code d'accès de la CAM.

5.1.1 Nature des identificateurs dans la CAM

Les principes fondateurs de la CAM sont d'une part la représentation des identificateurs de de Bruijn et d'autre part la théorie des catégories cartésiennes closes [6].

La compilation d'un identificateur sera intuitivement son code d'accès à la de Bruijn. D'ordinaire, ce code s'exprime souvent sous la forme d'un entier qui représente le nombre de liants séparant l'identificateur de l'abstraction à laquelle il est effectivement lié. Ici il est un peu différent, c'est une suite de i commandes `car` (où i est l'indice de de Bruijn associé à l'identificateur que l'on traduit en commençant à 0) suivi d'un `cdr` si la traduction réussit. On trouvera la justification de cette traduction dans [6]. L'indice de de Bruijn associé à une variable ML est aussi son adresse dans l'environnement Mini-ML (puisque l'on introduit une variable en tête de l'environnement Mini-ML à chaque déstructuration d'un liant). Lors de la compilation, c'est donc seulement la liste des variables dans l'ordre où elles ont été introduites dans l'environnement qui nous interesse et non pas l'environnement en entier ; un environnement ML est une liste d'association qui à un identificateur associe une valeur, on définie la structure `squelette` comme une liste d'identificateurs :

```
Inductive Set Squelette:=
nil_squelette:Squelette
|cons_squelette:Pat->Squelette->Squelette.
```

Le prédicat `Habite` suivant teste si un squelette "habite" bien un environnement, c'est à dire si il est bien la projection canonique sur la première composante de l'environnement :

```
Inductive Definition Habite :MLenv->Squelette->Prop:=

triv_habite:(Habite Enil nil_squelette)

|cons_habite:(x:Pat)(u:MLval)(e:MLenv)(s:Squelette)
              (Habite e s)->
               (Habite (Econs x u e) (cons_squelette x s)).
```

À un environnement, on ne peut donc associer qu'un squelette :

```
Lemma Habite_inject:
   (e:MLenv)(s1,s2:Squelette)
              (Habite e s1) ->
                (Habite e s2) ->
                   s1=s2.
```

La structure de `squelette` n'est pas utile per se, mais parce qu'elle simplifie la définition du prédicat `Access` défini ci-dessous. En effet, elle reflète le fait que seul l'ordre d'introduction des variables dans l'environnement Mini-ML est important pour traduire un identificateur en code d'accès de la CAM.

5.1.2 Traduction d'un identificateur

La traduction d'un identificateur en une suite de `car` suivi d'un `cdr` va alors de soi :

```
Inductive Definition Access:Pat->Squelette->Commande->Prop:=


    Rule1:(P:Pat)(s:Squelette)
                (Access P (cons_squelette P s) cdr)

  | Rule2:(P,T:Pat)(s:Squelette)(C:Commande)
               (~<Pat> P=T)->
                (Access P s C)->
                 (Access P (cons_squelette T s) (o car C)).
```

On peut facilement montrer qu'à un identificateur ne correspond qu'un seul code d'accès à la de Bruijn :

```
Lemma Access_inject :  (x:Pat)(s:Squelette)(C,C':Commande)
            (Access x s C')->
              (Access x s C)->
                       C=C'.
```

Maintenant que la traduction d'un identificateur est claire, on traite les autres constructions syntaxiques de Mini-ML.

5.1.3 La traduction

On définit la fonction de compilation dans un style de programme Prolog.

```
Inductive Definition Traduction:Squelette->MLexp->Commande->Prop:=

Bool_Trad:(b:bool)(S:Squelette)
              (Traduction S (Bool b) (quote (elem b)))

|Trad_num:(n:nat)(S:Squelette)
              (Traduction S (Num n) (quote (int n)))

|Trad_clos:(c:OP)(S:Squelette)
              (Traduction S (op c) (quote (def_op c)))

|Trad_var:(p:Pat)(S:Squelette)(C:Commande)
              (Access p S C)->(Traduction S (id p) C)

|Trad_ite:(S:Squelette)(E1,E2,E3:MLexp)(C1,C2,C3:Commande)
              (Traduction S E1 C1)->
                (Traduction S E2 C2)->
                  (Traduction S E3 C3)->
  (Traduction S (ite E1 E2 E3) (o push (o C1 (branch C2 C3))))

|Trad_pair:(S:Squelette)(E1,E2:MLexp)(C1,C2:Commande)
               (Traduction S E1 C1)->
                 (Traduction S E2 C2)->
  (Traduction S (mlpair E1 E2) (o push (o C1 (o swap (o C2 cons)))))

|Trad_app:(S:Squelette)(E1,E2:MLexp)(C1,C2:Commande)
               (Traduction S E1 C1)->
                 (Traduction S E2 C2)->
  (Traduction S (appl E1 E2) (o push (o C1 (o swap (o C2 (o cons app))))))

|Trad_let:(p:Pat)(S:Squelette)(E1,E2:MLexp)(C1,C2:Commande)
              (Traduction S E1 C1)->
                 (Traduction (cons_squelette p S) E2 C2)->
  (Traduction S (let' p E1 E2) (o push (o C1 (o cons C2))))

|Trad_let_rec:(p,x:Pat)(S:Squelette)(E,E2:MLexp)(C,C2:Commande)
 (Traduction (cons_squelette x (cons_squelette p S)) E C)->
                 (Traduction (cons_squelette p S) E2 C2)->
  (Traduction S (letrec p x E E2) (o push (o (cur_rec C) (o cons C2))))

|Trad_lambda:(S:Squelette)(p:Pat)(E:MLexp)(C:Commande)
               (Traduction (cons_squelette p S) E C)->
                 (Traduction S (lambda p E) (cur C)).
```

Encore une fois, on est fidèle à [9] et [16] si ce n'est pour la traduction de la récursion (règle `Trad_let_rec`). Le prédicat défini précédemment code une traduction déterministe :

```
Lemma Traduction_inject :
  (E:MLexp)(C,C':Commande)(s:Squelette)
              (Traduction s E C)->
                 (Traduction s E C')->
                           C=C'.
```

5.2 Equivalence entre domaines sources et domaines cibles

Avant de pouvoir énoncer la preuve de correction de la compilation, il nous faut encore définir une équivalence entre valeurs sémantiques de Mini-ML et de la CAM. En fait il nous faut aussi une équivalence entre environnements de Mini-ML et environnements de la CAM. De plus comme pour Mini-ML les valeurs et les environnements ont été définis de manière mutuellement récursive, on est obligé de définir deux prédicats `Equiv_val` et `Equiv_env` de façon récursive mutuelle. Le premier désigne l'équivalence des valeurs Mini-ML et CAM et le second l'équivalence des environnements des deux langages.

```
Mutual Inductive

Equiv_val:MLval->CSem_val->Prop:=

  Eqbool:(b:bool)(Equiv_val (boolean b) (val (elem b)))

 |Eqnum:(n:nat)(Equiv_val (num n) (val (int n)))

 |Eq_op:(c:OP)(Equiv_val (OP_clos c) (val (def_op c)))

 |Eqpair:(V1,V2:MLval)(Cval1,Cval2:CSem_val)
         (Equiv_val V1 Cval1)->
           (Equiv_val V2 Cval2)->
              (Equiv_val (valpair V1 V2) (Cam_pair Cval1 Cval2))

 |Eqclos:(p:Pat)(E:MLexp)(C:Commande)(e:MLenv)(CV:CSem_val)(s:Squelette)
              (Equiv_env e CV)->
                (Habite e s)->
                 (Traduction (cons_squelette p s) E C)->
                  (Equiv_val (Clos p E e) (Cam_clos C CV))

 |Eqclos_rec:(p,x:Pat)(E:MLexp)(e:MLenv)(C:Commande)(CV:CSem_val)
       (s:Squelette)
          (Equiv_env e CV)->
              (Habite e s)->
                (Traduction (cons_squelette x (cons_squelette p s)) E C)->
   (Equiv_val (Clos_rec x E p e) (Cam_clos_rec C CV))

with
Equiv_env:MLenv->CSem_val->Prop:=

  Eqenv1:(Equiv_env Enil Cam_nil)

 |Eqenv2:(p:Pat)(E:MLenv)(CV0:CSem_val)
            (Equiv_env E CV0)->
              (V:MLval)(CV:CSem_val)(Equiv_val V CV)->
                (Equiv_env (Econs p V E) (Cam_pair CV0 CV)).
```

Il faut remarquer ici que dans les constructeurs `Eqclos` et `Eqclos_rec`, il y a des occurrences de `Traduction`. Cela signifie donc que que l'on va prouver la correction de la `Traduction` vis à vis d'une équivalence qui utilise cette `Traduction`. Ceci parait absurde à première vue. En fait, nous prouvons au moins la correction de la traduction que sur les termes clos (car dans ce cas, aucune fermeture n'apparaît dans l'environnement initial d'évaluation).

Les équivalences entre valeurs sont naturelles. Ce qui est remarquable, c'est qu'il n'y a pas d'environnements à proprement parler pour la CAM. Ils sont codés à l'aide de la valeur vide et du constructeur de paires, comme ceci était précisé en (2.2).

5.3 Un peu de sucre syntaxique

Cete section illustre un nouveau comportement de l'utilisateur Coq dans la version 5.10. Jusqu'à maintenant, l'utilisateur du système Coq devait trouver des noms pour les objets qu'il axiomatisait, et il adoptait systématiquement une notation préfixe pas toujours très lisible. Les facilités syntaxiques offertes par la version 5.10 de l'aide à la démonstration Coq nous permettent d'adopter des notations proches de celles de G. Kahn ou de J. Despeyroux dans leurs présentations en sémantique naturelle.

Pour les prédicats représentant des sémantiques, on adopte une notations standard avec le symbole `|-` à l'aide de la commande `Grammar`. Ces nouvelle notations nous permettent d'écrire l'énoncé de notre preuve de manière élégante dans la section suivante.

```
Grammar command command1 :=
[command0($c) "|" "-" "id" command0($c0) ":" command0($c1)] ->
                        [$0 = <<(VAL_OF $c $c0 $c1)>>].

Grammar command command1 :=
[command0($c) "|" "-" "ml" command0($c0) ":" command0($c1)] ->
                        [$0 = <<(ML_DS $c $c0 $c1)>>].

Grammar command command1 :=
[command0($c) "|" "-" "cam" command0($c0) ":" command0($c1)] ->
                        [$0 = <<(CAM_DS $c $c0 $c1)>>].

Grammar command command1 :=
[command0($c) "|" "-" "trad" command0($c0) "->" command0($c1)] ->
                        [$0 = <<(Traduction $c $c0 $c1)>>].
```

Ces quatre phrases Coq permettent de définir une nouvelle syntaxe pour les prédicats `VAL_OF`, `ML_DS`, `CAM_DS` et `Traduction`. Maintenant, `s |-trad  C -> C1` sera parsé en `(Traduction s C C1)`. Il est possible de définir des conventions de *pretty − print* similaires [15].

Nous prenons de même des conventions agréables pour le tas de la machine abstraite catégorique.

```
Grammar command command1 :=
[command0($c) "@" command0($c0)] ->
                        [$0 = <<(ETcons $c $c0)>>].
```

On prend une notation infixe pour les équivalences entre valeurs (resp. environnements) de Mini-ml et de la machine catégorique abstraite : Attention néanmoins ! on ne désigne pas des égalités puisque déjà les deux membres n'ont pas le même type.

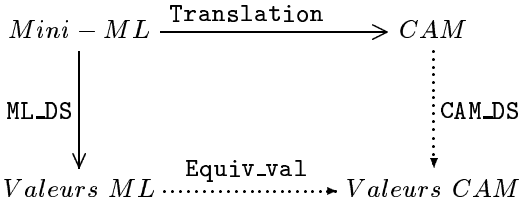
```
Grammar command command1 :=
[command0($c) "~" "val" command0($c0)] -> [$0 = <<(Equiv_val $c $c0)>>].

Grammar command command1 :=
[command0($c) "~" "env" command0($c0)] ->
                        [$0 = <<(Equiv_env $c $c0)>>].
```

5.4 La preuve finale

5.4.1 Enoncé

À l'aide des prédicats définis précédemment, on peut énoncer sous la forme d'un diagramme la preuve de correction :



5.4.2 Enoncé dans le système Coq

On peut maintenant donner l'énoncé de la preuve de correction dans Coq. Suivant le paradigme *proposition as type*, il se lit comme un énoncé mathématique où l'on interprête la flèche par l'implication, `(-:-)` par la quantification universelle et (Ex [-:-]()) par la quantification existentielle. On donne de plus une allure de "règle de déduction" à notre énoncé.

```
Lemma correctness :


                (E:MLexp)(e:MLenv)(V:MLval)
                     (e  |-ml  E : V)                      ->

                 (s:Squelette)(C:Commande)
                   (s   |-trad  E -> C)                    ->

                      (Habite e s)                         ->

                       (CV:CSem_val)
                       (e  ~env  CV)                       ->

(* ------------------------------------------------------------*)

                    ((Ex [CV1:CSem_val]
                      ((V  ~val  CV1)
        /\ (s:Etat)((CV @ s)  |-cam  C : (CV1 @ s))))).
```

Ce lemme a été démontré par récurrence sur le prédicat `ML_DS ((e  |-ml  E : V))`. Il faut noter que dans le cas où E est un identificateur, on a effectué une coupure et fait l'induction sur l'environnement Mini-ML `e`.

6 Comparaison avec d'autres preuves

Des preuves similaires ont déjà été effectuées avec d'autres langages, d'autres sémantiques et d'autres systèmes d'aide à la démonstration.

Une caractéristique importante de ces preuves est l'homogénéité avec laquelle les différentes sémantiques et translations sont décrites. B. Ciesielsky et M. Wand présentent leur preuve de manière entièrement équationnelle dans [2], F. Pfenning axiomatise dans [23] sa compilation et ses sémantiques opérationnelles

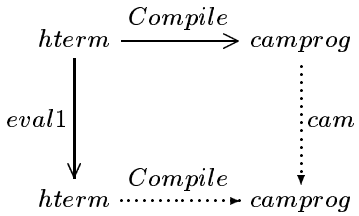
sous forme de systèmes de déduction. Nous-mêmes formalisons tous les protagonistes de notre preuve dans le style de la "sémantique naturelle".

B. Ciesielsky et M. Wand utilisent une sémantique dénotationnelle pour un fragment de Scheme (identificateurs, abstraction, application et addition). Tous les protagonistes de leur preuve sont présentés uniformément : sous forme d'ensemble d'équations. Ceci permet un traitement de la preuve par le système Isabelle-91 qui utilise essentiellement des tactiques de reécriture.

Pour effectuer la preuve de la compilation du λ-calcul en code CAM, F. Pfenning utilise une sémantique opérationnelle pour décrire l'évaluation du λ-calcul et de la CAM. Il réalise sa preuve en utilisant le langage Elf basé sur le calcul LF. Ainsi il utilise comme nous les analogies entre prédicats et types, et règles d'inférence et termes même si dans Elf il n'est pas possible de déclarer des types inductifs. Le fonctionnement de Elf peut être comparé à celui de Prolog et est basé sur un algorithme de résolution. Elf est donc essentiellement différent du système Coq. En particulier, les récurrences dans Elf ne sont pas effectuées par l'application d'un schéma d'élimination ; elles sont légitimées au niveau meta-théorique. La preuve que F. Pfenning réalise dans Elf est elle aussi très différente de la nôtre du point de vue axiomatique. En utilisant des sémantiques à la Plotkin, F. Pfenning choisit d'avoir le même domaine pour les expressions et les "valeurs". Il définit donc les catégories syntaxiques du λ-calcul et de la CAM (*htem* et *camprog*) et trois "familles de types" (correspondant à des types dépendants). La famille "$eval1 : hterm \rightarrow hterm \rightarrow type$" représente l'évaluation en "appel par valeurs". La famille "$Compile : hterm \rightarrow camprog \rightarrow type$" axiomatise la fonction de compilation. La famille de types "$cam : camprog \rightarrow camprog \rightarrow type$" axiomatise la sémantique correspondant à l'appel par valeurs pour la CAM.

Nôtre présentation axiomatique est très similaire à la manière dont les déclarations de ces familles dans Elf se font en utilisant des constantes (correspondants aux constructeurs de nos définitions de prédicats inductifs) dont les types sont l'axiomatisation de règles de déduction (comme nous l'avons fait pour le prédicat `Traduction` par exemple).

Mais la formulation de la preuve devient alors différente de la nôtre car il n'y a plus de catégories syntaxiques spéciales pour les valeurs et environnements des langages source et cible :



Dans Elf, il faut alors résoudre le but $correct : eval1\ E\ V \rightarrow compile\ E\ C \rightarrow cam\ C\ D \rightarrow compile\ V\ D \rightarrow type$ où E, V, C, D sont quantifiés universellement de manière implicite et où les types complets des arguments sont synthétisés automatiquement. Un tel lemme n'est pas prouvé directement dans Elf et l'utilisateur doit proposer des lemmes intermédiaires.

A. Felty et J. Seaman ont prouvé dans Coq (V5.8) la propriété connue sous le nom de *subject-reduction* et un théorème de normalisation pour le langage Lazy-PCF+Shar [14]. Naïvement, ce langage consiste en un λ-calcul simplement typé (à la Church) muni de constantes (entiers, booléens, successeur, prédecesseur et conditionnelle), d'un opérateur de point fixe (typé) et d'une construction permettant l'évaluation paresseuse et le partage. Ils utilisent une sémantique opérationelle pour décrire le comportement calculatoire du langage, et leur présentation est plus proche de celle de [22] que de la nôtre. Leur sémantique axiomatise une réduction et les valeurs appartiennent à la même catégorie syntaxique que les termes. Ils doivent donc gérer la substitution dans les termes de PCF.

Dans [23] F. Pfenning propose d'utiliser de la "syntaxe abstraite d'ordre supérieure" pour axiomatiser les langages dans LF. Ceci consiste à utiliser l'opérateur liant λ de LF pour axiomatiser l'abstraction du langage objet, le type du constructeur lambda du langage objet devient alors $(term \rightarrow term) \rightarrow term$. Il montre que cette approche simplifie l'axiomatisation de la substitution et les problèmes liés à la portée des identificateurs. Il est à noter que l'approche de nôtre preuve évite soigneusement ces problèmes.

J. Despeyroux, A. Hirschowitz et A. Felty, ont repris cette idée de F. Pfenning pour décrire des propriétés des sémantiques naturelles dans Coq cette fois [10] [11]. Dans [11], ils présentent leur axiomatisation de *Higher Order Abstract Syntax* dans le système Coq et prouvent *subject reduction* pour le λ-calcul.

7 Commentaires sur le système Coq

La preuve que nous présentons ici a été écrite dans plusieurs versions du système d'aide à la démonstration Coq. Depuis la première version [1], elle a subit à la fois des modifications conceptuelles et des modifications héritées de l'évolution de Coq. La version originelle a été écrite dans la V5.6. Une version modifiée a été écrite dans la V5.8. La version commentée ici a été réalisée dans la V5.10. L'évolution conceptuelle consiste en des modifications des sémantiques de [9] déjà décrite dans les sections précédentes. Elle a été effectuée dans la V5.8. Les modifications apportées avec l'évolution de Coq ont été réalisées dans la V5.10. Mis à part les facilités syntaxiques présentées dans la section 4, elles sont de deux natures. D'une part l'utilisation de tactiques dites d'inversion et d'autre part l'utilisation de définitions mutuellement récursives.

Les tactiques dites d'inversion du module `Inv` nous ont permis de supprimer les "inversions de prédicats inductifs". L'inversion d'un prédicat inductif du premier ordre (dont l'ensemble des règles sont du premier ordre) correspond au complété de Clark d'un programme Prolog. Le schéma d'élimination d'un prédicat `P` dans Coq permet de montrer que `P` implique son inverse. Ce principe peut se généraliser dans Coq à n'importe quel ordre encore que dans ce cas, il n'est pas toujours vrai qu'un prédicat implique son inverse. Dans les versions précédentes de Coq, pour inverser un prédicat `P`, il fallait écrire explicitement l'inverse `P_inv` et montrer la proposition `P->P_inv`. Ce travail n'est plus nécessaires dans la version 5.10 de Coq.

Les nouveaux outils de la version 5.10 permettent donc de faire mécaniquement des tâches qui s'avéraient longues et pénibles pour l'utilisateur : les inversions essentiellement, et la définition des structures récursives mutuelles. D'autre part, les facilités de *parsing* et de *pretty − printing* apportent beaucoup en lisibilité. Nous avons peu utilisé une autre facilité disponible dans la nouvelle version : les arguments implicites, nous renvoyons le lecteur au manuel de référence.

On peut tenter d'évaluer le temps mis pour réaliser une telle preuve. Il faut alors retrancher la familiarisation au système Coq, aux sémantiques opérationelles et l'apprentissage de la CAM du temps mis pour faire axiomatisations et preuves dans Coq. Lors du portage de cette preuve de compilation dans la dernière version du système Coq, il a fallut 1 mois à temps partiel pour rejouer toutes les séquences de tactiques de la preuve en remplaçant les anciennes inversions par l'utilisation des nouvelles tactiques d'inversion. Évidemment, dans ce portage, l'axiomatisation des différentes structures est conservée et le "cheminement" de la preuve est connu. On peut estimer que 2 à 3 mois sont nécessaire pour l'axiomatisation et la réalisation de la preuve dans Coq.

Le fichier de la preuve fait 1600 lignes. La preuve est compilée en un peu plus de 1h20 par un Sun 4 (sparc 2).

Conclusion

Dans ce papier nous avons présenté une preuve de correction de la compilation de Mini-ML en commandes CAM réalisée avec le système d'aide à la démonstration Coq.

Cette preuve a été faite relativement à des sémantiques naturelles des langages sources et cibles à l'instar de [9], ou [16].

Des expériences similaires ont été menées à la fois avec d'autre systèmes de preuves et avec d'autres sortes de sémantiques, on pourra consulter à ce propos [23]. Ici, le choix des langages et des sémantiques nous laissait espérer une mécanisation facile de la preuve pourtant complexe présentée dans [9].

Les sémantiques naturelles avec leur formulation "logique du premier ordre" s'axiomatisent "naturellement" comme des prédicats inductifs et se prêtent bien au traitement mécanique dans Coq . Cependant, l'utilisation de structures bouclées dans ces sémantiques a été un obstacle important dans notre axiomatisation.

Nous n'avons que partiellement mécanisé la preuve de [9] (2.7). Nous pensons que les structures quotients actuellement en cours d'implémentation permettrons de terminer cette mécanisation. Il s'agira alors de mécaniser les démonstrations proposées en appendice.

D'autre part la preuve que nous avons faite est facilement extensible à d'autres constructions syntaxiques des langages source et cible. Par exemple, une première version de [1] avait été réalisée sans constantes. L'ajout des constantes s'est avéré très simple : il s'agissait de rajouter un constructeur à la plus part des déclarations inductives et à rajouter une étape de récurrence dans la preuve finale. Une extension possible est la définition de *pattern* de Mini-ML plus générales que les simples variables de notre axiomatisation comme cela est présenté dans [16].

Remerciements

Je tiens à remercier B. Werner qui m'a beaucoup aidé à écrire une première version de cette preuve [1] et m'a initié à l'utilisation du système Coq. G. Huet m'a suggéré les raisons pour lesquelles le traitement en Mu-Prolog des sémantiques naturelles dans le système Typol ne pouvait pas être fidèlement représenté dans Coq. M. Mauny m'a été d'un précieux secours pour toute les questions que j'ai eu à me poser sur la CAM. Je remercie R. Fraer pour d'utiles commentaires sur une première version de ce rapport.

References

[1] S. Boutin *Verification en Coq d'un compilateur Mini-ML en CAM*. Rapport de Dea, Dea IMA, Septembre 1992.

[2] B.Ciesielski and M.Wand. *Using the Theorem Prover ISABELLE-91 to verify a simple Proof of Compiler Correctness*. College of Computer Science Notheastern University ; Boston ; 1991.

[3] D.Clement, J.Despeyroux, T.Despeyroux, G.Khan. *A simple Applicative Language : Mini-ML*. Rapport de recherche INRIA n^o 529 may 1986 also in the Proceedings of the ACM Conference on Lisp and Functional Programming, MIT, August 1986.

[4] T.Coquand. *Une Théorie des Constructions*. Thèse de troisième cycle ; Université Paris 7 ; 1985.

[5] T.Coquand, Ch Paulin. *Inductively Defined Types*. Proceedings of the International Conference on Computer Logic ; P.Martin-Löf,G.Mints (Eds.) ; Lecture Notes in Computer Science 417 ;1990.

[6] G. Cousineau, P.-L. Curien, M. Mauny. *The Categorical Abstract Machine*. Science of Computer Programming 8 ; 1987 ; 173-202 ; North-Holland.

[7] P. Crégut. *Machines à Environnements pour la Réduction Symbolique et l'Evaluation Partielle*. Thèse de doctorat présentée à Paris 7 en 1991.

[8] T. Despeyroux *Typol : a formalism to implement natural semantics* Rapport technique Inria n^o 94 ; March 1988.

[9] J. Despeyroux. *Proof of Translation in Natural Semantics*. Rapport de recherche INRIA n^o 017859 ;LCS 1986.

[10] J. Despeyroux, A. Hirschowitz. *Higher-order abstract syntax with induction in Coq* Proceedings of the International Conference on Logic Programming and Automated Reasoning 1994.

[11] J. Despeyroux, A. Hirschowitz, A. Felty *Higher-order abstract syntax in Coq* TLCA 95.

[12] G. Dowek. *Demonstration Automatique dans le Calcul des Constructions*. Thèse de doctorat présentée à Paris 7 en 1991.

[13] G. Dowek, A. Felty, H. Herbelin, G. Huet, Ch. Paulin, B. Werner. *The Coq Proof Assistant User's Guide*. Rapport technique INRIA n^o 134-Décembre 1991.

[14] A. Felty, J. Seaman *Proving Properties about a lazy functionnal language with the Coq Proof Development System* Draft Avril 1993.

[15] C. Cornes, J. Courant, J.C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, B. Werner. *The Coq Proof Assistant User's Guide Version 5.10*. Rapport technique Inria. À paraitre ;1995.

[16] G. Kahn. *Natural Semantics*. Proceedings of the Symposium on Theoretical Aspects of Computer Science, number 247 in LNCS, pages 22-39. Springer-Verlag, 1987.

[17] X. Leroy. *Typage Polymorphe d'un Langage Algorithmique*. Thèse de doctorat,Université Paris 7, 1992.

[18] M. Mauny. *Compilation des langages fonctionnels dans les combinateurs catégoriques. Application au langage ML.*. Thèse de doctorat,Université Paris 7, 1985.

[19] M.Mauny. *Calcul symbolique, Programmation et Genie logiciel*. Rapport technique Inria n^o 137, Juillet 1992.

[20] L. Naish. *Negation and Control in Prolog.* Lecture Notes in Computer Science , n^o 238, Springer-Verlag 1986.

[21] C. Paulin-Mohring *Inductive Definition in the System Coq - Rules and Properties.* Proceedings of the conference Typed Lambda Calculi and applications, March 1993. Also research report 92-49, LIP-ENS Lyon, December 1992.

[22] F. Pfenning, C. Elliott. *Higher-Order Abstract Syntax.* ACM SIGPLAN 1988, Conferrence on Programming Language Design and Implementation.

[23] F. Pfenning, J. Hannan *Compiler Verification in LF.* Seventh Annual Symposium on Logic in Computer Science, pages 407-418, 1992

[24] Gordon D. Plotkin. *A Structural Approach to Operational Semantics.* Unpublished Course Notes, Université de Aarhus,Denmark, 1981.

[25] B. Werner. *Une théorie des constructions inductives* Thèse de doctorat présentée à Paris 7 en 1994.

Equivalence entre ML et ML'

Nous prouvons maintenant que le diagramme suivant commute :

$$\begin{array}{ccc} Mini-ML & \xrightarrow{ID} & Mini-ML \\ \Big\downarrow{\scriptstyle (ML')} & & \Big\downarrow{\scriptstyle (ML)} \\ V'_{ML} & \xrightarrow{t_{ml}} & V_{ML} \end{array}$$

Rappelons que ML' = ML(1),ML(2),....,ML(12) et que ML = ML' - ML(12) + ML(12") + ML(10bis). On note $\vdash_\star$ la dérivation de ML' et $\vdash$ celle de ML.

Définition 1

$$\begin{gathered} t_{ml}(true) = true \\ t_{ml}(false) = false \\ t_{ml}(number(n)) = number(n) \\ t_{ml}([\![\lambda x.E;\rho]\!]) = [\![\lambda x.E;\rho]\!] \\ t_{ml}([\![\lambda x.E;(\rho_1 = P \leftarrow [\![\lambda x.E;\rho_1]\!].\rho)]\!]) = [\![P;\lambda x.E;t_{ml}(\rho)]\!]_{rec} \\ t_{ml}(\emptyset) = \emptyset \\ t_{ml}(P \leftarrow V.\rho) = P \leftarrow t_{ml}(V).t_{ml}(\rho) \\ t_{ml}(\rho_1 = P \leftarrow [\![\lambda x.E;\rho_1]\!].\rho) = P \leftarrow [\![P;\lambda x.E;t_{ml}(\rho)]\!]_{rec}.t_{ml}(\rho) \end{gathered}$$

Lemme 1 *Étant donné un environnement ρ une expression de Mini-ML E et une valeur Mini-ML V,*

$$\rho \vdash_\star E : V \Rightarrow t_{ml}(\rho) \vdash E : t_{ml}(V)$$

Ceci se prouve par induction sur la dérivation $\rho \vdash_\star E : V$. Tous les cas découlent simplement des hypothèses de récurrence, nous donnons quand même l'analyse pour la récursion. Si la dernière règle utilisée est

$$\frac{\rho_1 = P \leftarrow [\![\lambda P.E;\rho_1]\!].\rho \quad , \quad \rho 1 \vdash_\star E_2 : \alpha}{\rho \vdash_\star \text{letrec } P = \lambda x.E \text{ in } E_2 : \alpha}$$

Alors par hypothèse de récurrence, on a

$$t_{ml}(\rho_1 \ = \ P \ \leftarrow \ [\![\lambda P.E; \rho_1]\!].\rho) \vdash E_2 : t(\alpha)$$

Ce qui se simplifie en

$$P \leftarrow [\![P; \lambda x.E; t_{ml}(\rho)]\!]_{rec}.t_{ml}(\rho) \vdash E_2 : t(\alpha)$$

On applique alors ML(12") pour obtenir

$$t_m l(\rho) \vdash \text{letrec } P = \lambda x.E \text{ in } E_2 : t(\alpha)$$

CQFD

Equivalence entre CAM et CAM'

Nous prouvons maintenant que le diagramme suivant commute :

$$\begin{array}{ccc} \text{CAM} & \xrightarrow{T_{cam}} & \text{CAM'} \\ (\text{CAM})\downarrow & & \downarrow(\text{CAM'}) \\ V_{\text{CAM}} & \xrightarrow{t_{cam}} & V_{\text{CAM'}} \end{array}$$

Rappelons que CAM' = CAM(1),CAM(2),....,CAM(17)
et que CAM = CAM(1),CAM(2)...CAM(16),CAM(18),CAM(16bis).
On note $\rightarrow^\star$ la dérivation de CAM' et $\rightarrow$ celle de CAM.
Contrairement à la translation de ML' vers ML, ici la sémantique et le langage sont différents, même si les différences sont minimes. T_{cam} est définie par induction sur la forme des instructions de la CAM.

Définition 2

$$\begin{gathered} T_{cam}(cur_{rec}(C)) = push; quote(\rho 1); cons; push; cur(T_{cam}(C)); swap; rplac; cdr \\ T_{cam}(cur(C)) = cur(T_{cam}(C)) \\ T_{cam}(C1; C2) = T_{cam}(C1); T_{cam}(C2) \\ T_{cam}(C) = C \quad sinon \end{gathered}$$

L'équivalence t_{cam} est définie par induction sur la structure des valeurs et environnements de CAM.

Définition 3

$$\begin{gathered} t_{cam}(bool(true)) = bool(true) \\ t_{cam}(bool(false)) = bool(false) \\ t_{cam}(int(n)) = int(n) \\ t_{cam}([\![C; \rho]\!]) = [\![T_{cam}(C); t_{cam}(\rho)]\!] \\ t_{cam}(\emptyset) = \emptyset \\ t_{cam}(\alpha, \beta) = (t_{cam}(\alpha), t_{cam}(\beta)) \\ t_{cam}([\![C; \rho]\!]_{rec}) = (\rho_1 = [\![T_{cam}(C); (t_{cam}(\rho), \rho_1)]\!]) \end{gathered}$$

On étend t_{cam} aux états de CAM (à l'aide de la fonction "map")

Lemme 2 *Étant donné des état s et s' de la CAM et C une suite d'instructions.*

$$s \rightarrow_C^\star s' \Rightarrow t_{cam}(s) \rightarrow_{T_{cam}(C)} t_{cam}(s')$$

Ceci se prouve par induction sur la dérivation $s \rightarrow_C^\star s'$. Nous n'analysons que les cas des règles CAM(18),CAM(16bis), les autres cas se déduisant simplement des hypothèses de récurrence.

1. Si l'on a

$$\rho.s \rightarrow_{cur_{rec}(COM)} [\![COM;\rho]\!]_{rec}.s$$

Alors considérant $T_{cam}(cur_{rec}(COM))$, on a

$$\begin{array}{rl}
t_{cam}(\rho).s \rightarrow_{push} t_{cam}(\rho).t_{cam}(\rho).s & CAM(10)\\
\rightarrow_{quote(\rho_1)} \rho_1.t_{cam}(\rho).s & CAM(6)\\
\rightarrow_{cons} (t_{cam}(\rho),\rho_1).s & CAM(9)\\
\rightarrow_{push} (t_{cam}(\rho),\rho_1).(t_{am}(\rho),\rho_1).s & CAM(10)\\
\rightarrow_{cur(T_{cam}(C))} [\![C;(t_{cam}(\rho),\rho_1)]\!].(t_{cam}(\rho),\rho_1).s & CAM(15)\\
\rightarrow_{swap} (t_{cam}(\rho),\rho_1).[\![C;(t_{cam}(\rho),\rho_1)]\!].s & CAM(11)\\
\rightarrow_{rplac} (t_{cam}(\rho),(\rho_1 = ([\![C;(t_{cam}(\rho),\rho_1)]\!]))).s & CAM(17)
\end{array}$$

et comme de plus par définition de t_{cam} on a

$$(\rho_1 = ([\![C;(t_{cam}(\rho),\rho_1)]\!])) \;=\; t_{cam}([\![C;t_{cam}(\rho)]\!]_{rec}$$

On obtient exactement

$$s \rightarrow^{\star}_{cur_{rec}(C)} s' \Rightarrow t_{cam}(s) \rightarrow_{T_{cam}(cur_{rec}(C))} t_{cam}(s')$$

2. si la dernière règle de la dérivation est :

$$\frac{((\rho,[\![C;\rho]\!]_{rec}),\alpha).s \rightarrow_C s_1}{([\![C;\rho]\!]_{rec},\alpha).s \rightarrow_{app} s_1}$$

Alors par hypothèse de récurrence, on a

$$(t_{cam}(\rho),(\rho_1 = [\![T_{cam}(C);(t_{cam}(\rho),\rho_1)]\!])).t_{cam}(s) \rightarrow_{T_{cam}(C)} t_{cam}(s_1)$$

D'autre part

$$t_{cam}(([\![C;\rho]\!]_{rec},\alpha)) = ((\rho_1 = [\![T_{cam}(C);(t_{cam}(\rho),\rho_1)]\!]),t_{cam}(\alpha))$$

En appliquant CAM(16) il vient

$$((\rho_1 = [\![T_{cam}(C);(t_{cam}(\rho),\rho_1)]\!]),t_{cam}(\alpha)).t_{cam}(s) \rightarrow_{app} t_{cam}(s_1)$$

avec $app = T_{cam}(app)$

CQFD

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399