5 Leçon 5

5.1 Sémantique opérationnelle grands pas de mini-Caml

Tout ceci est bel et bon, et modulo quelques concessions à l'informatique (l'usage de mémoires, la manie obsessionnelle de vouloir des points fixes de toute fonction que l'on puisse écrire dans le langage) on a réussi à définir une sémantique au goût mathématique de mini-Caml.

Mais ce n'est pas tout : il faudra exécuter les programmes mini-Caml, et pour ceci nous cherchons à les traduire en assembleur, langage que la machine comprend.

Nous allons faire ceci par étapes. La première étape va être de définir une nouvelle sémantique du langage mini-Caml, plus proche de la machine, et nous montrerons qu'il existe une fonction de traduction de la précédente vers la nouvelle, qui est correcte au sens où un diagramme ressemblant à (1) commute.

Nous allons aussi en profiter pour introduire une nouvelle forme de sémantique, dite *opérationnelle*, comme celle de l'assembleur en section 3.1.4, mais un peu plus riche. À première vue, il n'y aura pas grande différence. Une sémantique opérationnelle permet de dériver des *jugements* à l'aide d'un ensemble bien défini de *règles*. Ici, nous utiliserons des jugements de la forme

$$\rho, \mu \vdash M \Rightarrow \mu', V$$

qui signifient que dans l'environnement $\rho \in Env$, en partant de la mémoire $\mu \in Mem$, l'exécution de M peut terminer en retournant $\mu' \in Mem$ comme nouvel état de la mémoire, et la valeur V. Ceci est à rapprocher de l'affirmation $(\mu', V) = [\![M]\!]_{\text{caml}}\,\rho\mu$, et, jusqu'ici, nous dirons que c'est affaire de goût de préférer l'une ou l'autre présentation. Une première différence, cependant, c'est que la sémantique dénotationnelle était par essence déterministe : M a exactement une valeur dans un environnement et une mémoire donnés. Une sémantique opérationnelle permet de dire que M s'évalue, de façon non-déterministe, à une valeur V_1 ou à une autre valeur V_2.

5.1.1 Clôtures

L'essentiel de la nouvelle sémantique, cependant, est que nous allons tenter de rendre la notion de fonction plus concrète. Dans la sémantique de la section 3.2, la valeur d'une fonction est une vraie fonction mathématique. Mais ceci ne se représente pas sur un ordinateur ! Comme on l'a vu, la notion la plus proche que l'on ait en assembleur est la notion d'adresse d'exécution d'un morceau de code.

Nous allons donc donner une nouvelle sémantique de mini-Caml dans laquelle la valeur d'une fonction `fun` $x \to M$ sera, à peu de choses près, une adresse dans un morceau de code. Comme nous n'en sommes pas encore réellement à compiler en assembleur, nous allons estimer que l'expression syntaxique `fun` $x \to M$ est elle-même un symbole représentant l'adresse d'un code à exécuter, et la valeur de `fun` $x \to M$ sera tout simplement l'expression syntaxique `fun` $x \to M$ elle-même. (Ceci nécessite de changer le domaine Val des valeurs de sorte qu'il contienne une copie isomorphe de l'ensemble des programmes du langage.)

Ceci, malheureusement, ne fonctionne pas. Considérons l'exemple :

```
letrec f = fun x ->
              let g = fun y -> x+y
              in g;;
let a = f 3 4;;
```

Selon cette explication, la valeur de `g` dans la définition de `f` sera juste l'expression syntaxique `fun y -> x+y`. Mais alors, que vaut `a` ? Ce serait la valeur de `(fun y -> x+y) 4`, soit `x+4`, oui, mais pour quelle valeur de `x` ? La sémantique de la section 3.2 dit à la place que, dans la mesure où `f` est appelée avec l'argument 3, elle retourne la fonction qui à y associe $3 + y$, puis cette fonction est appliquée à 4, retournant 7. Donc il ne suffit pas de retourner le *code* `fun y -> x+y`, il faut aussi retourner un environnement qui nous informe que ce code doit être compris avec `y` valant 3.

Un couple formé d'une expression de la forme $\texttt{fun}\ x \to M$ et d'un environnement ρ est traditionnellement appelé une *clôture*. En pratique, on ne sait pas représenter un environnement complet, qui associe une valeur à chaque variable dans Var, un ensemble infini. Heureusement, on n'a besoin que des variables qui apparaissent *libres* dans $\texttt{fun}\ x \to M$. (Encore une fois, on ignore les instructions arithmétiques ici.)

Définition 6 (Variables libres) *L'ensemble* $\mathrm{fv}(M)$ *des* variables libres *d'une expression mini-Caml M est défini par récurrence structurelle sur M par :*

$$\begin{array}{lll}
\mathrm{fv}(x) = x & \mathrm{fv}(MN) = \mathrm{fv}(M) \cup \mathrm{fv}(N) & \mathrm{fv}(\texttt{fun}\ x \to M) = \mathrm{fv}(M) \setminus \{x\} \\
 & & \mathrm{fv}(\texttt{let}\ x = M\ \texttt{in}\ N) = \mathrm{fv}(M) \cup (\mathrm{fv}(N) \setminus \{x\}) \\
\mathrm{fv}(\texttt{ref}\ M) = \mathrm{fv}(M) & \mathrm{fv}(!M) = \mathrm{fv}(M) & \mathrm{fv}(M{:=}N) = \mathrm{fv}(M) \cup \mathrm{fv}(N) \\
 & \mathrm{fv}(\texttt{proj}_i M) = \mathrm{fv}(M) & \mathrm{fv}((M_1, \ldots, M_n)) = \mathrm{fv}(M_1) \cup \ldots \cup \mathrm{fv}(M_n) \\
 & \mathrm{fv}(M; N) = \mathrm{fv}(M) \cup \mathrm{fv}(N) & \mathrm{fv}(\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P) = \mathrm{fv}(M) \cup \mathrm{fv}(N) \cup \mathrm{fv}(P)
\end{array}$$

▸ **EXERCICE 5.1**

Montrer que la sémantique dénotationnelle d'une expression mini-Caml ne dépend que des valeurs de ses variables libres. Autrement dit, pour tout expression M mini-Caml, montrer que si ρ et ρ' sont deux environnements tels que $\rho(x) = \rho'(x)$ pour tout $x \in \mathrm{fv}(M)$, alors $[\![M]\!]_{\mathrm{caml}}\, \rho\mu = [\![M]\!]_{\mathrm{caml}}\, \rho'\mu$.

Par l'exercice 5.1, on peut se contenter d'apparier une fonction syntaxique avec un *environnement fini*, c'est-à-dire une fonction ϱ partielle des variables vers les valeurs, de domaine fini. On en vient à définir les valeurs des fonctions sous forme de *clôtures* :

Définition 7 (Clôture simple) *Une* clôture simple *est un triplet $\langle x, M, \varrho\rangle$, où $x \in Var$, M est une expression mini-Caml, et $\varrho \in Var \to_{fin} Val'$ est un environnement fini tel que* $\mathrm{fv}(M) \setminus \{x\} \subseteq \mathrm{dom}\ \varrho$. *On notera $Clos_s$ l'ensemble de toutes les clôtures simples.*

Cette définition est à compléter par une définition de l'ensemble Val' des valeurs que nous utiliserons dans la définition de notre nouvelle sémantique. Pour les différencier de l'ensemble Val des valeurs que nous utilisions jusqu'ici, nous appellerons Val' l'ensemble des valeurs *concrètes*, et Val celui des valeurs *abstraites*. Si la valeur abstraite d'une fonction est une fonction continue, la valeur concrète de la même fonction sera une clôture : les valeurs concrètes ne sont pas les valeurs abstraites, et Val' ne peut donc pas être le même ensemble que Val.

Avant de définit l'espace des valeurs concrètes, nous devons prévoir de pouvoir définir des fonctions récursives, c'est-à-dire celles définies par `letrec` (que nous verrons en section 5.1.4). Or, si l'environnement ϱ envoie le symbole de fonction f vers la clôture $\langle x, M, \varrho' \rangle$, le corps M de la clôture ne peut pas faire référence à f, c'est-à-dire à la même clôture. (Dans des réalisations informatiques pratiques, on peut s'arranger pour que ϱ' fasse "pointer" le symbole f vers la clôture $\langle x, M, \varrho' \rangle$, créant ainsi un cycle. Ceci serait maladroit à réaliser dans la description, encore très mathématique, que nous nous apprêtons à effectuer.) Nous définissons donc :

Définition 8 (Clôture récursive) *Une* clôture récursive *est un quadruplet* $\texttt{fix}\langle f, x, M, \varrho \rangle$*, où* $f, x \in Var$*,* M *est une expression mini-Caml, et* $\varrho \in Var \to_{fin} Val'$ *est un environnement fini tel que* $\mathrm{fv}(M) \setminus \{f, x\} \subseteq \mathrm{dom}\ \varrho$*. On notera* $Clos_r$ *l'ensemble de toutes les clôtures récursives.*

L'idée est que $\texttt{fix}\langle f, x, M, \varrho \rangle$ dénote la fonction qui à x associe M, étant entendu que dans le calcul de M, toute référence à f dénote cette même fonction. On notera que $Clos_r$ et $Clos_s$ sont disjoints.

Définition 9 (Valeurs concrètes) *L'ensemble des* clôtures *est* $Clos = Clos_s \cup Clos_r$.

L'ensemble Val' *des* valeurs concrètes *est le plus petit ensemble tel que*

$$Val' \supseteq Clos + Addr + Val'^{*} + \mathbb{Z} + \mathbb{B} \tag{18}$$

où Val'^{*} *dénotent l'ensemble des suites finies d'éléments de* Val'*, et* $+$ *est la somme disjointe usuelle.*

Il faut bien remarquer qu'ici Val' et $Clos$ sont des *ensembles*, pas des cpo. Il se trouve en effet que l'inégalité (18) a une plus petite solution en tant qu'ensemble :

▸ **EXERCICE 5.2**

Soit Val' l'ensemble de tous les arbres finis construits comme suit. Un sommet d'un tel arbre est étiqueté par l'un des symboles `CLOS` $(x, M, [x_1, \ldots, x_n])$ ou `FIXCLOS` $(f, x, M, [x_1, \ldots, x_n])$ (où $x \in Var$, M est une expression mini-Caml, x_1, …, x_n sont n variables distinctes et contenant au moins toutes les variables libres de M sauf possiblement x, et possiblement f dans le second cas), `ADDR` (a) (où $a \in Addr$), `SUITE`, `Z` (n) (où $n \in \mathbb{Z}$), ou `B` (n) (où $n \in \mathbb{B}$). Un sommet étiqueté `CLOS` $(x, M, [x_1, \ldots, x_n])$, resp. `FIXCLOS` $(f, x, M, [x_1, \ldots, x_n])$ a n fils : si ces n fils représentent des éléments $v_1, \ldots, v_n$ de Val', alors l'arbre a ce sommet représente la clôture $\langle x, M, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \rangle$, resp. $\texttt{fix}\langle f, x, M, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \rangle$. Un sommet étiqueté `SUITE` a une suite finie quelconque de fils. Les autres sommets n'ont pas de fils. Montrer que Val' obéit à l'équation (18), modulo un isomorphisme facile à trouver.

La nouvelle sémantique que nous définissons pour mini-Caml se démarque de celle de la section 3.2 essentiellement en ce qui concerne les fonctions, et on définira donc :

$$\frac{x \in \text{dom } \varrho}{\varrho, \mu \vdash x \Rightarrow \mu, \varrho(x)} \ (Var)$$

$$\frac{\varrho, \mu \vdash M \Rightarrow \mu', \overbrace{\langle x, M', \varrho' \rangle}^{\text{clôture simple}} \qquad \varrho, \mu' \vdash N \Rightarrow \mu'', V \qquad \varrho'[x \mapsto V], \mu'' \vdash M' \Rightarrow \mu''', V'}{\varrho, \mu \vdash MN \Rightarrow \mu''', V'} \ (App)$$

$$\frac{}{\varrho, \mu \vdash \texttt{fun } x \to M \Rightarrow \mu, \quad \underbrace{\langle x, M, \varrho \rangle}_{\text{clôture simple}}} \ (Fun)$$

À ce point, il est bon de digérer un peu ces définitions. Nous les commentons en section 5.1.2. Nous devrons encore traiter des fonctions définies par `letrec`, pour lesquelles nous avons introduit les clôtures récursives : nous reléguons ce cas à la section 5.1.4.

5.1.2 Appel gauche-droite, par valeur, par nécessité, par référence

Contrairement à la sémantique opérationnelle que nous avons définie pour l'assembleur (section 3.1.4), la règle (App) notamment a des prémisses. Ceci signifie que, pour évaluer l'application MN dans un environnement fini ϱ, en partant d'une mémoire μ (conclusion), il suffit de :

- évaluer M dans ϱ en partant de μ, et si le résultat existe et donne une nouvelle mémoire μ', et une valeur qui est une clôture $\langle x, M', \varrho' \rangle$, alors :
- évaluer l'argument N dans ϱ en partant de cette dernière mémoire μ' ; si le résultat existe et fournit une troisième mémoire μ'', et une valeur V, alors :
- évaluer le corps M' de la clôture dans l'environnement fini ϱ' de la clôture, modifié de telle sorte que x vaille la valeur V de l'argument.

Si cette dernière étape réussit et fournit une dernière mémoire μ''' et une valeur V', alors μ''', V est le résultat de l'évaluation de MN.

On a donc défini une règle d'évaluation *de la gauche vers la droite* : d'abord M est évalué en une clôture, ensuite son argument est calculé. On aurait aussi bien pu calculer de la droite vers la gauche. Peu de langages le font. Une exception notable est Caml.

▸ **EXERCICE 5.3**
Écrire une règle d'application modifiée où l'argument N serait évalué avant la fonction M.

Une autre caractéristique de la règle (App) est qu'elle fonctionne en *appel par valeur* : la valeur V de l'argument N est calculée *avant* d'être passée au corps M' de la fonction. D'autres langages, dits *paresseux*, comme Miranda ou Haskell, utilisent la notion d'appel *par nécessité*, où l'argument N n'est tout simplement pas évalué, et est passé tel quel à M'. En appel par nécessité, c'est l'analogue de la règle (Var) qui cherchera à effectivement calculer la valeur des variables. L'avantage est que l'argument N ne sera évalué que si le corps M' de la fonction requiert effectivement la valeur de la variable x pour effectuer son calcul.

Le langage C fonctionne, comme Caml ou mini-Caml, en appel par valeur. Mais d'autres langages encore utilisent l'*appel par référence*. Par exemple, en C++, on peut écrire :

```
int f (int &x)
{
  x = x+1;
  return x+2;
}
```

La caractéristique essentielle ici est la déclaration `int &x`, qui déclare que le paramètre entier `x` est passé non pas par valeur comme d'habitude (on aurait juste écrit `int x`), mais par référence. Autrement dit, si l'on écrit :

```
  int n = 3;
  int m = f (n);
```

au lieu de calculer `n`$= 3$, d'appeler `f` en recopiant l'entier 3 dans une nouvelle variable `x` propre à `f`, puis de retourner le résultat 6 et de le stocker dans `m`, comme cela se passerait en appel par valeur, ici l'appel `f (n)` va passer directement l'adresse de `n` au code de `f`, de sorte que la variable `x` de `f` soit *exactement* au même endroit que la variable `n`. Ceci a comme conséquence que l'instruction `x = x+1;` de la première ligne du code de `f` va en fait incrémenter l'entier `n`.

On aurait pu écrire le même code en C pur, c'est-à-dire en appel par valeur, en utilisant les pointeurs :

```
int f (int *xp)
{
  *xp = *xp+1;
  return *xp+2;
}

int n = 3;
int m = f (&n);
```

A noter que l'opération `&n` récupère l'adresse de la variable `n` en mémoire (on pourra comparer avec l'instruction `leal`, cf. section 3.1.4), et `*xp` récupère le contenu de l'adresse (pointeur) stockée dans `xp` (on comparera avec le mode d'adressage indirect de l'assembleur).

5.1.3 Autres règles

Le reste des constructions de mini-Caml reçoit une sémantique sans surprise, inspirée des règles de la sémantique dénotationnelle de la section 4.1.2. La construction `let` se décrit comme en (6) :

$$\frac{\varrho, \mu \vdash M \Rightarrow \mu', V \qquad \varrho[x \mapsto V], \mu' \vdash N \Rightarrow \mu'', V'}{\varrho, \mu \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N \Rightarrow \mu'', V'}\ (Let)$$

En utilisant une stratégie d'allocation mémoire $alloc$, on obtient les analogues de (7), (8), (9). Il est à noter que dans la sémantique opérationnelle de cette section, nous supposerons implicitement que μ, μ', … varient parmi l'ensemble Mem' des *mémoires concrètes*, à savoir des fonctions partielles de domaine fini de $Addr$ vers Val'.

$$\frac{\varrho, \mu \vdash M \Rightarrow \mu', V \quad a = alloc(\mathrm{dom}\ \mu') \neq \bot}{\varrho, \mu \vdash \mathtt{ref}\ M \Rightarrow \mu'[a \mapsto V], a}\ (Ref)$$

$$\frac{\varrho, \mu \vdash M \Rightarrow \mu', a \quad a \in \mathrm{dom}\ \mu'}{\varrho, \mu \vdash !M \Rightarrow \mu', \mu'(a)}\ (!)$$

$$\frac{\varrho, \mu \vdash M \Rightarrow \mu', a \quad \varrho, \mu' \vdash N \Rightarrow \mu'', V \quad a \in \mathrm{dom}\ \mu''}{\mu''[a \mapsto V], \epsilon}\ (:=)$$

▸ **EXERCICE 5.4**
Traduire les règles (10), (11), (12), (13) en sémantique opérationnelle. Les n-uplets $(M_1, \ldots, M_n)$ sont-ils évalués de gauche à droite, de droite à gauche, ou dans un autre ordre encore ? Pourquoi ?

5.1.4 Sémantique opérationnelle des programmes

Comme dans la sémantique dénotationnelle, nous devons donner la sémantique des programmes. Pour ceci, nous définirons une nouvelle forme de jugement

$$\varrho, \mu \vdash d \Rightarrow \varrho', \mu'$$

exprimant que l'évaluation de la ou les déclarations d en partant d'un environnement fini ϱ et d'une mémoire μ produit le nouvel environnement fini ϱ' et la nouvelle mémoire μ'. Le cas du `let` est simple, et est similaire à la règle (15) :

$$\frac{\varrho, \mu \vdash M \Rightarrow \mu', V}{\varrho, \mu \vdash \mathtt{let}\ x = M;; \Rightarrow \varrho[x \mapsto V], \mu'}\ (Let;;)$$

Le cas du `letrec` est bien entendu plus compliqué. Notons que `letrec` est la seule construction de mini-Caml permettant d'écrire des boucles (récursion, `while`). Si la nouveauté de la sémantique que nous écrivons était les clôtures dans le cas des expressions, dans le cas des programmes, ce sera le traitement des définitions récursives. C'est la raison pour laquelle nous avons introduit les clôtures récursives en définition 8 :

$$\frac{}{\varrho, \mu \vdash \mathtt{letrec}\ f = \mathtt{fun}\ x \to M;; \Rightarrow \varrho[f \mapsto \underbrace{\mathtt{fix}\langle f, x, M, \varrho\rangle}_{\text{clôture récursive}}], \mu}\ (Letrec;;)$$

À part pour l'utilisation d'une clôture récursive, cette règle serait pratiquement la combinaison d'une application de $(Let;;)$ et de (Fun). Il nous reste à définir comment l'on évalue l'application d'une clôture récursive à un argument. C'est la règle $(FixApp)$ ci-dessous ; rappelons en effet que la règle (App) ne traitait que du cas de l'application d'une clôture *simple*.

$$\frac{\varrho, \mu \vdash M \Rightarrow \mu', \overbrace{\texttt{fix}\langle f, x, M', \varrho'\rangle}^{\text{clôture récursive}} \quad \varrho, \mu' \vdash N \Rightarrow \mu'', V \quad \varrho'[f \mapsto \texttt{fix}\langle f, x, M', \varrho'\rangle][x \mapsto V], \mu'' \vdash M' \Rightarrow \mu''', V'}{\varrho, \mu \vdash MN \Rightarrow \mu''', V'} \ (FixApp)$$

Notons que la différence avec la règle (App) est que, pour évaluer M', on se place dans un environnement fini dans lequel f est lié à la clôture récursive $\texttt{fix}\langle f, x, M', \varrho'\rangle$, de nouveau. Nous illustrerons en passant l'usage de cette règle dans la suite.

5.1.5 Arbres de dérivations, récurrence sur les dérivations

Soit $\mathcal{R}$ un ensemble de règles, par exemple, l'ensemble des règles définies dans cette leçon 5.

Définition 10 (Dérivation) *Pour tout jugement J, l'ensemble $Der(\mathcal{R}, J)$ de toutes les* dérivations *de J à partir de $\mathcal{R}$ est par définition le plus petit ensemble tel que, pour toute instance de règle de conclusion J :*

$$\frac{J_1 \ldots J_n}{J} \ (R)$$

pour toutes dérivations π_1 de J_1, …, π_n de J_n à partir de $\mathcal{R}$, alors

$$\frac{\begin{matrix} \vdots\, \pi_1 & & \vdots\, \pi_n \\ J_1 & \ldots & J_n \end{matrix}}{J} \ (R) \tag{19}$$

est une dérivation de J à partir de $\mathcal{R}$.

L'ensemble $Der(\mathcal{R})$ des dérivations *à partir de $\mathcal{R}$ est l'union sur tous les jugements J de $Der(\mathcal{R}, J)$.*

Une *instance* d'une règle est, informellement, ce que l'on obtient à partir de l'énoncé d'une règle en remplaçant chaque méta-variable (μ, ϱ, M, etc.) par les objets correspondants (ici, une mémoire, un environnement fini, un terme, etc.). La définition 10 définit les dérivations comme des *arbres finis* : la dérivation (19) se lit comme l'arbre dont la racine est étiquetée par le jugement J, et ayant n fils qui sont des arbres π_1, …, π_n de racines étiquetées par J_1, …, J_n respectivement. (Note aux informaticiens : ces arbres sont écrits à l'envers, avec la racine en bas. Note aux autres : ces arbres sont écrits à l'endroit, avec la racine en bas.)

On omettra souvent de dire "à partir de $\mathcal{R}$" lorsque le contexte définira $\mathcal{R}$ clairement. Ici, $\mathcal{R}$ sera l'ensemble des règles écrites en section 5.1.

Voici par exemple une dérivation du fait que $\texttt{fun}\ x \to x$ s'évalue à une clôture simple dans un environnement vide et une mémoire vide :

$$\frac{}{[], [] \vdash \texttt{fun}\ x \to x \Rightarrow [], \langle x, x, []\rangle} \ (Fun)$$

Cette dérivation ne consiste qu'en l'application d'une instance de la règle (Fun). C'est une dérivation complète, car (Fun) n'a aucune prémisse. En général, les premières règles appliquées dans une dérivation (celles qui sont tout en haut) sont des règles sans prémisses. En l'occurrence, les règles sans prémisses sont (Var), (Fun), $(Letrec;;)$. Ce sont donc les seules règles qui permettent de démarrer une dérivation.

Un peu plus compliqué, voici une dérivation montrant que, si l'on part d'une mémoire où l'entier 1 est stocké à l'adresse $\texttt{a}_1$, d'un environnement fini envoyant x vers 2 et m vers $\texttt{a}_1$, et que l'on évalue l'expression $\texttt{let}\ y = !m\ \texttt{in}\ (m{:=}x; y)$:

$$\dfrac{\dfrac{\dfrac{}{\begin{array}{l}[x \mapsto 2, m \mapsto \texttt{a}_1],\\ [\texttt{a}_1 \mapsto 1]\\ \vdash\\ m \Rightarrow [\texttt{a}_1 \mapsto 1], \texttt{a}_1\end{array}}(Var)}{\begin{array}{l}[x \mapsto 2, m \mapsto \texttt{a}_1],\\ [\texttt{a}_1 \mapsto 1]\\ \vdash\\ !m \Rightarrow [\texttt{a}_1 \mapsto 1], 1\end{array}}(!) \qquad \dfrac{\dfrac{\dfrac{}{\begin{array}{l}[x \mapsto 2, m \mapsto \texttt{a}_1,\\ \quad y \mapsto 1],\\ [\texttt{a}_1 \mapsto 1]\\ \vdash\\ m \Rightarrow [\texttt{a}_1 \mapsto 1], \texttt{a}_1\end{array}}(Var) \quad \dfrac{}{\begin{array}{l}[x \mapsto 2, m \mapsto \texttt{a}_1,\\ \quad y \mapsto 1],\\ [\texttt{a}_1 \mapsto 1]\\ \vdash\\ x \Rightarrow [\texttt{a}_1 \mapsto 1], 2\end{array}}(Var)}{\begin{array}{l}[x \mapsto 2, m \mapsto \texttt{a}_1, y \mapsto 1],\\ [\texttt{a}_1 \mapsto 1]\\ \vdash\\ m{:=}x \Rightarrow [\texttt{a}_1 \mapsto 2], \epsilon\end{array}}(:=) \quad \dfrac{}{\begin{array}{l}[x \mapsto 2, m \mapsto \texttt{a}_1,\\ \quad y \mapsto 1],\\ [\texttt{a}_1 \mapsto 2]\\ \vdash\\ y \Rightarrow [\texttt{a}_1 \mapsto 2], 1\end{array}}(Var)}{\begin{array}{l}[x \mapsto 2, m \mapsto \texttt{a}_1, y \mapsto 1], [\texttt{a}_1 \mapsto 1]\\ \vdash\\ (m{:=}x; y) \Rightarrow [\texttt{a}_1 \mapsto 2], 1\end{array}}(;)}{[x \mapsto 2, m \mapsto \texttt{a}_1], [\texttt{a}_1 \mapsto 1] \vdash \texttt{let}\ y = !m\ \texttt{in}\ (m{:=}x; y) \Rightarrow [\texttt{a}_1 \mapsto 2], 1}(Let)$$

On a ici utilisé une règle $(;)$ non définie explicitement (voir exercice 5.4). A noter que chaque règle s'applique bien, au sens où les *conditions de bord* sont vérifiées. La condition de bord de la règle (Var) est que la variable à évaluer est dans le domaine de l'environnement fini à gauche du symbole thèse ($\vdash$). Nous n'avons pas inclus les conditions de bord dans l'écriture des règles ci-dessus, par manque de place. Par exemple, on aurait formellement dû écrire $m \in \mathrm{dom}\ [x \mapsto 2, m \mapsto \texttt{a}_1]$ en haut de l'instance la plus à gauche de (Var).

Le fait que les dérivations soient des arbres finis, ou de façon équivalente que $Der(\mathcal{R})$ soit défini comme l'ensemble le plus petit vérifiant les conditions de la définition 10, est caractérisé par l'existence de *principes de récurrence*.

Le principe de démonstration par *récurrence structurelle* sur les dérivations est le suivant : pour montrer qu'une propriété P est vraie de toutes les dérivations, il suffit de vérifier que, pour chaque règle

$$\frac{J_1 \ldots J_n}{J}\,(R)$$

de $\mathcal{R}$, si toute dérivation de J_1, …, J_n vérifie P, alors

$$\frac{\begin{array}{c}\vdots\ \pi_1\\ J_1\end{array} \quad \ldots \quad \begin{array}{c}\vdots\ \pi_n\\ J_n\end{array}}{J}\,(R)$$

vérifie P. Les cas de base de la récurrence sont ceux des règles R pour lesquelles $n = 0$. Les autres sont les cas de récurrence. Voici un exemple d'application de ce principe.

Théorème 14 (Déterminisme) *La sémantique opérationnelle de mini-Caml est* déterministe*, c'est-à-dire que, pour tout environnement fini ϱ, pour toute mémoire concrète μ, pour toute expression M mini-Caml, il existe au plus une mémoire concrète μ' et une valeur concrète V telles que $\varrho, \mu \vdash M \Rightarrow \mu', V$ soit dérivable. (On dit qu'un jugement est* dérivable *si et seulement s'il en existe une dérivation.)*

Démonstration. Soient π_1 et π_2 deux dérivations dérivant des jugements de la forme demandée, autrement dit

$$\begin{array}{c} \vdots\, \pi_1 \\ \varrho, \mu \vdash M \Rightarrow \mu'_1, V_1 \end{array} \qquad \begin{array}{c} \vdots\, \pi_2 \\ \varrho, \mu \vdash M \Rightarrow \mu'_2, V_2 \end{array}$$

Nous allons montrer que $\pi_1 = \pi_2$, ce qui implique $\mu'_1 = \mu'_2$ et $V_1 = V_2$. Ceci se fera par récurrence structurelle sur π_1. Il y a autant de cas que de règles qui ont pu être appliquées pour conclure π_1 (nous ne traiterons que le cas des règles définies explicitement dans ce cours, à l'exclusion des règles de l'exercice 5.4 et de la règle $(FixApp)$ à venir ; nous laissons en exercice le soin de vérifier que le théorème reste vrai si on ajoute ces règles, ou au besoin de les corriger).

(Var) La dérivation π_1 se termine par (Var), c'est-à-dire est de la forme

$$\frac{}{\varrho, \mu \vdash x \Rightarrow \mu, \varrho(x)}$$

avec $x \in \mathrm{dom}\ \varrho$. En particulier, M vaut x. La seule règle applicable pour conclure π_2 est donc (Var) aussi, et donc π_2 est exactement π_1. En particulier $\mu'_1 = \mu = \mu'_2$, $V_1 = \varrho(x) = V_2$.

(App) C'est le cas le plus compliqué ; π_1 est de la forme :

$$\frac{\begin{array}{c} \vdots\, \pi'_1 \\ \varrho, \mu \vdash M_1 \Rightarrow \mu'_1, \langle x_1, M'_1, \varrho'_1 \rangle \end{array} \quad \begin{array}{c} \vdots\, \pi''_1 \\ \varrho, \mu'_1 \vdash N \Rightarrow \mu''_1, V_1 \end{array} \quad \begin{array}{c} \vdots\, \pi'''_1 \\ \varrho'_1[x_1 \mapsto V_1], \mu''_1 \vdash M' \Rightarrow \mu'''_1, V'_1 \end{array}}{\varrho, \mu \vdash M_1 N \Rightarrow \mu'''_1, V'_1} (App)$$

En particulier, $M = M_1 N$, et la seule règle applicable en fin de π_2 est donc encore (App). Autrement dit, π_2 est de la forme :

$$\frac{\begin{array}{c} \vdots\, \pi'_2 \\ \varrho, \mu \vdash M_1 \Rightarrow \mu'_2, \langle x_2, M'_2, \varrho'_2 \rangle \end{array} \quad \begin{array}{c} \vdots\, \pi''_2 \\ \varrho, \mu'_2 \vdash N \Rightarrow \mu''_2, V_2 \end{array} \quad \begin{array}{c} \vdots\, \pi'''_2 \\ \varrho'_2[x_2 \mapsto V_2], \mu''_2 \vdash M' \Rightarrow \mu'''_2, V'_2 \end{array}}{\varrho, \mu \vdash M_1 N \Rightarrow \mu'''_2, V'_2} (App)$$

L'hypothèse de récurrence appliquée à π'_1 et π'_2 nous permet de déduire que $\pi'_1 = \pi'_2$, en particulier $\mu'_1 = \mu'_2$, $x_1 = x_2$, $M'_1 = M'_2$, $\varrho'_1 = \varrho'_2$. Comme $\mu'_1 = \mu'_2$, l'hypothèse de récurrence est applicable à π''_1 et à π''_2, donc $\pi''_1 = \pi''_2$. En particulier, $\mu''_1 = \mu''_2$ et

$V_1 = V_2$. Il s'ensuit que $\varrho'_1[x_1 \mapsto V_1] = \varrho'_2[x_2 \mapsto V_2]$, et comme de plus $\mu''_1 = \mu''_2$, on peut appliquer l'hypothèse de récurrence une troisième fois et conclure $\pi'''_1 = \pi'''_2$. Ceci implique en particulier $\mu'''_1 = \mu'''_2$ et $V'_1 = V'_2$. Ces derniers résultats plus les trois égalités $\pi'_1 = \pi'_2$, $\pi''_1 = \pi''_2$, et $\pi'''_1 = \pi'''_2$ impliquent que $\pi_1 = \pi_2$.

(Fun) Le cas de (Fun) est similaire à celui de (Var).

Autres Les cas (Let), (Ref), $(!)$, $(:=)$ sont similaires au cas (App). À noter que dans le cas (Ref), tout dépend du fait que $alloc$ est une fonction d'allocation : le théorème serait faux si on avait décidé (comme il est l'usage !) que la nouvelle adresse a était une adresse quelconque hors du domaine de μ'. Il n'y aurait alors aucune raison que les deux mémoires que π_1 et π_2 produisent soient identiques. (Point technique : elles serait identiques à permutation des adresses près, cependant.) $\square$

5.1.6 Règles dérivées

Nous avons dit au début de ce cours que l'on pouvait coder la boucle `while` en CaML au moyen du `letrec`, au sens où `while (b) e` pouvait se coder sous la forme `boucle ()`, où l'on a défini :

```
letrec boucle x =
       if b
          then (e; boucle ())
       else ();;
```

Une fois évaluée cette dernière définition par la règle $(Letrec;;)$, l'environnement fini courant ϱ_1 lie `boucle` à la clôture récursive $\mathtt{fix}\langle\mathtt{boucle}, \mathtt{x}, \mathtt{if\ b\ then\ (e;\ boucle\ x)\ else\ ()}, \varrho_0\rangle$, pour un certain environnement fini ϱ_0, et coincide avec ϱ_0 sur toutes les autres variables.

Définissons `while (b) e` comme étant l'expression `boucle ()`. Si `b` s'évalue à faux, c'est-à-dire si l'on dispose d'une dérivation π de $\varrho_1[\mathtt{x} \mapsto \epsilon], \mu \vdash \mathtt{b} \Rightarrow \mu', 0$, la seule dérivation d'un jugement de la forme $\varrho_1, \mu \vdash \mathtt{boucle}() \Rightarrow \ldots$ est :

$$\dfrac{\dfrac{}{\begin{array}{l}\varrho_1, \mu \\ \vdash \\ \mathtt{boucle} \Rightarrow \mu, \\ \mathtt{fix}\langle\mathtt{boucle}, \mathtt{x}, \\ \ \mathtt{if\ b} \\ \ \ \mathtt{then\ (e;\ boucle\ ())} \\ \ \mathtt{else\ ()}, \\ \ \varrho_0\rangle\end{array}}\,(Var) \qquad \dfrac{}{\varrho_1, \mu \vdash () \Rightarrow \mu, \epsilon}\,(()) \qquad \dfrac{\dfrac{\vdots\,\pi}{\begin{array}{l}\varrho_2, \mu \\ \vdash \\ \mathtt{b} \Rightarrow \mu', 0\end{array}} \quad \dfrac{}{\begin{array}{l}\varrho_2, \mu' \\ \vdash \\ () \Rightarrow \mu', \epsilon\end{array}}\,(())}{\begin{array}{l}\varrho_2, \mu \\ \vdash \\ \ \mathtt{if\ b} \\ \ \ \mathtt{then\ (e;\ boucle\ ())} \\ \ \mathtt{else\ ()} \\ \Rightarrow \mu', \epsilon\end{array}}\,(If_0)}{\varrho_1, \mu \vdash \mathtt{boucle}() \Rightarrow \mu', \epsilon}\,(FixApp) \qquad (20)$$

Encore une fois, nous n'avons pas défini les règles $(())$ et (If_0) (voir exercice 5.4 ; nous supposons que les règles (If_0) et (If_1) définissant la sémantique opérationnelle du `if` évaluent la branche `then` si la condition booléenne s'évalue à 1, la branche `else` si la condition booléenne s'évalue à 0, et qu'il n'y a aucune règle applicable dans les autres cas). Nous avons utilisé le nom ϱ_2 pour l'environnement fini $\varrho_0[\texttt{boucle} \mapsto \texttt{fix}\langle\texttt{boucle}, \texttt{x}, \texttt{if b then (e; boucle x) else ()}, \varrho_0\rangle][\texttt{x} \mapsto \epsilon]$. Notons que cette expression se simplifie en $\varrho_2 = \varrho_1[\texttt{x} \mapsto \epsilon]$.

Si `b` s'évalue à vrai, c'est-à-dire si l'on dispose d'une dérivation π de $\varrho_1[\texttt{x} \mapsto \epsilon], \mu \vdash \texttt{b} \Rightarrow \mu', 1$, la seule dérivation d'un jugement de la forme $\varrho_1, \mu \vdash \texttt{boucle}() \Rightarrow \ldots$ est la suivante, où nécessairement l'on dispose d'une dérivation π' d'un jugement de la forme $\varrho_2, \mu' \vdash \texttt{e} \Rightarrow \mu'', V'$ et d'une dérivation π'' d'un jugement de la forme $\varrho_2, \mu'' \vdash \texttt{boucle}() \Rightarrow \mu''', V$:

$$\dfrac{\dfrac{}{\begin{array}{l}\varrho_1, \mu \\ \vdash \\ \texttt{boucle} \Rightarrow \mu, \\ \texttt{fix}\langle\texttt{boucle}, \texttt{x}, \\ \ \texttt{if b} \\ \ \ \texttt{then (e;} \\ \ \ \ \texttt{boucle ())} \\ \ \texttt{else ()}, \\ \ \varrho_0\rangle\end{array}}(Var) \qquad \dfrac{}{\varrho_1, \mu \vdash () \Rightarrow \mu, \epsilon}(()) \qquad \dfrac{\begin{array}{c}\vdots\,\pi \\ \varrho_2, \mu \\ \vdash \\ \texttt{b} \Rightarrow \mu', 1\end{array} \quad \dfrac{\begin{array}{c}\vdots\,\pi' \\ \varrho_2, \mu' \\ \vdash \\ \texttt{e} \Rightarrow \mu'', V'\end{array} \quad \begin{array}{c}\vdots\,\pi'' \\ \varrho_2, \mu'' \\ \vdash \\ \texttt{boucle}() \Rightarrow \mu''', V\end{array}}{\begin{array}{l}\varrho_2, \mu' \\ \vdash \\ \texttt{(e; boucle ())} \Rightarrow \mu', V\end{array}}(())}{\begin{array}{l}\varrho_2, \mu \\ \vdash \\ \texttt{if b} \\ \ \texttt{then (e;} \\ \ \ \texttt{boucle ())} \\ \texttt{else ()} \\ \Rightarrow \mu', V\end{array}}(If_0)}{\varrho_1, \mu \vdash \texttt{boucle}() \Rightarrow \mu', V}(FixApp) \tag{21}$$

Il est donc équivalent de définir la sémantique de `while (b) e` par les deux règles :

$$\dfrac{\varrho_1, \mu \vdash \texttt{b} \Rightarrow \mu', 0}{\varrho_1, \mu \vdash \texttt{while (b) e} \Rightarrow \mu', \epsilon}(While_0)$$

et

$$\dfrac{\begin{array}{c}\varrho_1, \mu \vdash \texttt{b} \Rightarrow \mu', 1 \\ \varrho_1, \mu' \vdash \texttt{e} \Rightarrow \mu'', V' \quad \varrho_1, \mu'' \vdash \texttt{while (b) e} \Rightarrow \mu', V\end{array}}{\varrho_1, \mu \vdash \texttt{while (b) e} \Rightarrow \mu', V}(While_1)$$

(Techniquement, ceci suppose que $\varrho_2 = \varrho_1$. Nous réglons ce point technique plus bas, voir les exercices 5.5, 5.6.) La démonstration de l'équivalence procède comme suit. D'abord, la direction facile : si l'on a utilisé la règle $(While_0)$ ou $(While_1)$, on peut remplacer l'instance de l'une ou l'autre règle par le morceau de dérivation (20) ou (21) respectivement. Dans l'autre direction, on a vu que toute dérivation d'un jugement de la forme $\varrho_1, \mu \vdash \texttt{boucle}() \Rightarrow \ldots$ était nécessairement de la forme (20) ou (21) (par exemple, il est facile de voir que la dernière règle appliquée est nécessairement $(FixApp)$, et l'on continue de la même façon, en construisant la forme des dernières règles appliquées depuis le bas de la dérivation) ; et l'on peut alors remplacer une fin de dérivation de forme (20) par une instance de $(While_0)$, et une fin de dérivation de forme (21) par une instance de $(While_1)$.

La règle $(While_0)$ exprime que si `b` vaut faux, alors on sort de la boucle tout de suite, avec une mémoire résultant de l'évaluation du seul booléen `b`. La règle $(While_1)$ peut sembler paradoxale, vu que pour évaluer `while (b) e` (en conclusion), la troisième prémisse demande à évaluer... `while (b) e`. Mais notons que cette troisième prémisse demande à évaluer `while (b) e` à partir d'une mémoire μ'' différente de la mémoire μ de départ ; notamment μ'' est le résultat des mises à jour de la mémoire effectuées lors de l'évaluation de `b` et de `e` (dans cet ordre). Ce sont, au passage, les règles classiques de sémantique opérationnelle des boucles `while` dans les langages impératifs comme C. On a donc en particulier montré (modulo quelques points techniques de détail) que la boucle `while (b) e` pouvait se coder via un `letrec`, au sens où la sémantique opérationnelle est préservée.

La règle $(While_1)$ illustre un point important : dans une dérivation, il n'est pas nécessaire que les prémisses portent sur des termes à évaluer plus petits que ceux de la conclusion. On obtiendra une dérivation bien formée à partir du moment où la règle $(While_1)$ n'est appliquée qu'un nombre fini de fois, c'est-à-dire si et seulement si la boucle `while (b) e` *termine*.

▸ **EXERCICE 5.5**

Démontrer que, si x n'est pas libre dans M, alors on peut dériver $\varrho[x \mapsto V_1], \mu \vdash M \Rightarrow \mu', V$ *si et seulement si* on peut dériver $\varrho, \mu \vdash M \Rightarrow \mu', V$. Indication : étant donné une dérivation π de l'un des deux jugements, construire une dérivation de l'autre par récurrence structurelle sur π. Faites bien apparaître où vous avez utilisé l'hypothèse selon laquelle x n'est pas libre dans M ; en particulier, dans le cas de quelle règle ceci est-il important ?

Un langage ayant la propriété de l'exercice 5.5, c'est-à-dire où la sémantique de toute expression M ne dépend que des valeurs des variables libres dans M, est appelé un langage à *liaison lexicale*. C'est le cas de Caml, de C, de Pascal, de Java. Ce n'est pas le cas de la plupart des langages de la famille Lisp : en prenant l'exemple d'Emacs-Lisp, le programme

```
(setq x 2)   ; définition de x comme valant 2,
(defun f (y) (+ x y))  ; on définit f comme la fonction
                       ; qui ajoute x à son argument,
(defun g (x) (f x))    ; apparemment g est juste f...
(g 3)                  ; et pourtant (g 3) retourne 6?
```

alors que l'équivalent Caml :

```
let x=2;;
let f = fun y -> x+y;;
let g = fun x -> f x;;
g 3;;
```

retourne 5, conformément aux sémantiques présentées ici. La raison pour laquelle `(g 3)` retourne 6 en Emacs-Lisp est que `f` est définie comme la fonction qui à y associe $x + y$, avec x la valeur courante *au moment de l'appel de* `f` de `x` : or l'appel de `g` sur l'argument `3` a lié (temporairement) `x` à 3, changeant ainsi la sémantique de `f` du même coup. Dans un langage à liaison lexicale, la différence est que x serait la valeur *au moment de la définition de* `f`.

▸ **EXERCICE 5.6**

En utilisant l'exercice 5.5, montrez formellement l'équivalence entre la définition de `while (b) e` comme une abréviation de `boucle ()`, dans tout environnement liant `boucle` à la clôture récursive $\texttt{fix}\langle\texttt{boucle}, \texttt{x}, \texttt{if b then (e; boucle x) else ()}, \varrho_0\rangle$; et la définition de `while (b) e` via les règles $(While_0)$ et $(While_1)$. Ceci nécessite des hypothèses supplémentaires, à savoir que `x` ne doit être libre ni dans `b` ni dans `e`. Donnez un contre-exemple à l'équivalence si `x` est libre dans `e`.

5.1.7 Correction de la sémantique concrète par rapport à l'abstraite

Nous avons maintenant deux sémantiques, l'une dénotationnelle, l'autre opérationnelle, de mini-Caml, et il serait nécessaire de montrer qu'elles sont équivalentes. Nous avions suggéré qu'une telle propriété de correction serait matérialisée par un diagramme de la forme (1), c'est-à-dire de façon abstraite un diagrame de la forme

$$\begin{tikzcd} \mathbf{Caml} \arrow[r, "c"] \arrow[dr, "[\![\_]\!]_{\mathrm{caml}}"'] & \mathbf{Asm} \arrow[d, "[\![\_]\!]_{\mathrm{asm}}"] \\ & V \end{tikzcd}$$

Outre le fait que nous ne disposons pas (encore ?) d'une fonction $[\![\_]\!]_{\mathrm{asm}}$, nous devons réviser nos ambitions pour la raison que nos deux sémantiques n'évaluent pas les expressions mini-Caml dans le même domaine de valeurs ! C'est un phénomène général : non seulement un compilateur traduit des programmes (mini-Caml vers assembleur dans l'exemple du diagramme 1), mais il suppose aussi un *changement de représentation* des valeurs. La propriété de correction que nous chercherons à montrer est donc plutôt un diagramme commutatif de la forme :

$$\begin{tikzcd} \mathbf{Caml} \arrow[r, "c"] \arrow[d, "[\![\_]\!]_{\mathrm{caml}}"'] & \mathbf{Asm} \arrow[d, "[\![\_]\!]_{\mathrm{asm}}"] \\ V & V' \arrow[l, "\alpha"'] \end{tikzcd} \tag{22}$$

où α est une fonction d'*abstraction*, qui à tout élément concret, de V', associe sa représentation au niveau abstrait, c'est-à-dire dans V.

Ces considérations sont essentiellement une affaire de style, et ne sont pas fondamentales, comme le suggère l'exercice suivant.

▸ **EXERCICE 5.7**

En changeant au besoin la définition de $[\![\_]\!]_{\mathrm{asm}}$, pourquoi le diagramme (1) contient-il le diagramme (22) comme cas particulier ? En définissant α de façon adéquate, pourquoi le diagramme (22) contient-il le diagramme (1) comme cas particulier ?

Dans le cas qui nous occupe de la correction de la sémantique opérationnelle de mini-Caml par rapport à sa sémantique dénotationnelle, on pourrait donc penser qu'il suffirait de montrer la commutativité d'un diagramme de la forme

$$\begin{tikzcd} \mathbf{Caml} \arrow[r, "c"] \arrow[d, "[\![\_]\!]_{\text{caml}}"'] & \mathbf{Caml} \arrow[d, "[\![\_]\!]_{\text{caml}'}"] \\ V & V' \arrow[l, "\alpha"'] \end{tikzcd} \tag{23}$$

Le fait que la sémantique concrète (de droite) ne soit pas écrite en style dénotationnel n'est pas un problème. Il suffit de la définir par :

$$[\![M]\!]_{\text{caml}'}\ \varrho\mu = \{(\mu', V) | \varrho, \mu \vdash M \Rightarrow \mu', V\}$$

Notez que, par le théorème 14, l'ensemble du côté droit a au plus un élément. Il en a exactement un si et seulement si l'évaluation de M termine dans l'environnement fini ϱ, en partant de la mémoire concrète μ.

La fonction de compilation c est aussi très simple : on n'a pas du tout modifié les programmes, c'est la fonction identité de $\mathbf{Caml}$ dans $\mathbf{Caml}$.

La vraie difficulté est de trouver la fonction d'abstraction α qui doit relier valeurs concrètes (dans Val') aux valeurs abstraites correspondantes (dans Val).

Définition 11 *Soit $\alpha_0 : Val' \to Val$ la fonction définie comme suit.*
- *$\alpha_0(v) = v$ si $v \in Addr + \mathbb{Z} + \mathbb{B}$;*
- *$\alpha_0(v_1 \cdot \ldots \cdot v_n) = \alpha_0(v_1) \cdot \ldots \cdot \alpha_0(v_n)$;*
- *$\alpha_0\langle x, M, \varrho\rangle = \phi$, où ϕ est l'unique fonction continue de $Mem \times Val$ dans $(Mem \times Val)_\perp$ telle que $\phi(\mu, V) = [\![M]\!]_{\text{caml}}(\rho[x \mapsto V])\mu$, et où ρ est n'importe quel environnement tel que, pour tout $y \in fv(M) \setminus \{x\}$, $\rho(y) = \alpha_0(\varrho(y))$; (remarquez bien la différence de typographie entre ρ et ϱ ;)*
- *$\alpha_0(\texttt{fix}\langle f, x, M, \varrho\rangle) = lfp(F)$, où F est la fonction continue de $Fun = [Mem \times Val \to (Mem \times Val)_\perp]$ dans Fun qui à tout $\phi \in Fun$ associe l'unique $\phi' \in Fun$ tel que $(\mu, \phi') = [\![\texttt{fun}\ x \to M;;]\!]_{\text{caml}}(\rho[f \mapsto \phi])\mu$, où ρ est n'importe quel environnement tel que, pour tout $y \in fv(M) \setminus \{f, x\}$, $\rho(y) = \alpha_0(\varrho(y))$ (remarquez bien la différence de typographie entre ρ et ϱ).*

Cette définition est correcte. Notamment, les deux derniers cas ne dépendent pas de quel environnement ρ est réellement choisi, pourvu qu'il associe à chaque variable y dans $\text{fv}(M) \setminus \{x\}$, resp. $\text{fv}(M) \setminus \{f, x\}$, la valeur $\alpha_0(\varrho(y))$: c'est une conséquence de l'exercice 5.1. Les propriétés de continuité nécessaires à vérifier sont conséquences du théorème 13.

Intuitivement, cette définition de α_0 dit que toute adresse, tout entier, tout booléen se représente lui-même (premier cas), qu'un n-uplet représente le n-uplets de ce que ses composantes représentent (deuxième cas), enfin qu'une clôture simple ou récursive représente la fonction continue décrite par la fonction $[\![\_]\!]_{\text{caml}}$ de sémantique dénotationnelle. On rappelle que lfp est l'opérateur de plus petit point fixe.

Le diagramme (23) n'est cependant pas le bon. Si l'on s'en tient à la lettre, le domaine V dans lequel $[\![\_]\!]_{\mathrm{caml}}$ prend ses valeurs n'est pas Val, mais $[Env \times Mem \to (Mem \times Val)_\perp]$. De même, V' devrait être l'ensemble de toutes les fonctions de $Env' \times Mem'$ vers $\mathbb{P}_1(Mem' \times Val')$, où Env' est l'ensemble des environnements finis (à valeurs dans Val'), Mem' est l'ensemble des mémoires concrètes, et $\mathbb{P}_1(E)$ dénote l'ensemble des parties de cardinal au plus un de E. Or trouver une fonction d'abstraction α convenable de $V' = Env' \times Mem' \to \mathbb{P}_1(Mem' \times Val')$ vers $V = [Env \times Mem \to (Mem \times Val)_\perp]$ ne semble pas une tâche facile.

On va court-circuiter la difficulté en changeant l'énoncé du diagramme de correction :

Définition 12 (Correction forte) *On dira que la sémantique* $[\![\_]\!]_{\mathrm{caml}'}$ *est* correcte *par rapport à la sémantique* $[\![\_]\!]_{\mathrm{caml}}$*, pour le compilateur* $c : \mathbf{Caml} \to \mathbf{Caml}$*, si et seulement si :*

- *pour toute expression mini-Caml* M*,*
- *pour tout environnement fini* ϱ *tel que* $\mathrm{fv}(M) \subseteq \mathrm{dom}\ \varrho$*, et tout environnement* ρ *tel que* $\rho(y) = \alpha_0(\varrho(y))$ *pour tout variable* $y \in \mathrm{fv}(M)$*,*
- *pour toute mémoire concrète* μ*, pour toute mémoire* $\tilde{\mu}$ *telles que* $\mathrm{dom}\ \mu \subseteq \mathrm{dom}\ \tilde{\mu}$ *et* $\tilde{\mu}(a) = \alpha_0(\mu(a))$ *pour tout* $a \in \mathrm{dom}\ \mu$*,*

alors :

- *ou bien* $[\![c(M)]\!]_{\mathrm{caml}'}\ \varrho\mu = \emptyset$ *et* $[\![M]\!]_{\mathrm{caml}}\ \rho\tilde{\mu} = \perp$ *(non-terminaison) ;*
- *ou bien* $[\![c(M)]\!]_{\mathrm{caml}'}\ \varrho\mu$ *est de la forme* $\{(\mu', v)\}$*,* $[\![M]\!]_{\mathrm{caml}}\ \rho\mu$ *est de la forme* $(\tilde{\mu}', V) \neq \perp$*,* $V = \alpha_0(v)$*,* $\mathrm{dom}\ \mu' \subseteq \mathrm{dom}\ \tilde{\mu}'$ *et* $\tilde{\mu}'(a) = \alpha_0(\mu'(a))$ *pour tout* $a \in \mathrm{dom}\ \mu'$*.*

Bien sûr, ou bien $[\![c(M)]\!]_{\mathrm{caml}'}\ \varrho\mu = \emptyset$ ou bien $[\![c(M)]\!]_{\mathrm{caml}'}\ \varrho\mu$ est de la forme $\{(\mu', v)\}$. La correction signifie donc que, dans le premier cas, où M ne termine pas dans la sémantique concrète, alors M ne "termine pas" non plus dans la sémantique abstraite (sa valeur est $\perp$) ; et dans le second cas, où M calcule une mémoire et une valeur concrètes dans la sémantique concrète, alors ce que M calcule dans la sémantique abstraite est la mémoire abstraite et la valeur abstraite correspondantes.

D'un point de vue technique, la seconde propriété se démontre par récurrence structurelle sur les dérivations. En effet, dire que $[\![c(M)]\!]_{\mathrm{caml}'}\ \varrho\mu$ est de la forme $\{(\mu', v)\}$, c'est dire que l'on a une dérivation π de $\varrho, \mu \vdash c(M) \Rightarrow \mu', v$, sur laquelle on peut effectuer une récurrence structurelle pour montrer que $[\![M]\!]_{\mathrm{caml}}\ \rho\mu$ est de la forme $(\tilde{\mu}', V) \neq \perp$, avec $V = \alpha_0(v)$, $\mathrm{dom}\ \mu' \subseteq \mathrm{dom}\ \tilde{\mu}'$ et $\tilde{\mu}'(a) = \alpha_0(\mu'(a))$ pour tout $a \in \mathrm{dom}\ \mu'$. C'est ce que nous ferons dans le théorème 15 ci-dessous.

La première propriété, qui énonce la préservation de la terminaison, peut se reformuler comme suit, par contraposée : si $[\![M]\!]_{\mathrm{caml}}\ \rho\tilde{\mu} \neq \perp$, alors il existe une dérivation d'un jugement $\varrho\mu \vdash M \Rightarrow \mu', v$ pour une certaine mémoire concrète μ' et une certaine valeur concrète v. Intuitivement, on pourrait construire cette dérivation à partir du bas, en regardant la valeur abstraite de M à chaque étape pour déduire quelle valeur concrète pourrait lui correspondre, mais ceci ne fonctionne pas : il nous faut montrer que ce processus de construction de dérivation produit une dérivation, c'est-à-dire un arbre *fini* ; autrement dit, que la construction de la dérivation finit par s'arrêter. Or il n'y aucune raison que ceci se produise. Mais je n'ai aucun contre-exemple, ni aucune preuve pour l'instant.

On a en tout cas le résultat, plus faible :

Théorème 15 (Correction faible) *La sémantique* $[\![\_]\!]_{\text{caml}'}$ *est* faiblement correcte *par rapport à la sémantique* $[\![\_]\!]_{\text{caml}}$ *pour le compilateur* $\text{id} : \textbf{Caml} \to \textbf{Caml}$ *: pour toute expression* M*, si* $[\![c(M)]\!]_{\text{caml}'} \varrho\mu = \{(\mu', v)\}$*, c'est-à-dire s'il existe une dérivation* π *de* $\varrho, \mu \vdash M \Rightarrow \mu', v$*, alors* $[\![M]\!]_{\text{caml}} \rho\tilde{\mu}$ *est de la forme* $(\tilde{\mu}', V) \neq \bot$*, avec* $V = \alpha_0(v)$*,* $\text{dom } \mu' \subseteq \text{dom } \tilde{\mu}'$ *et* $\tilde{\mu}'(a) = \alpha_0(\mu'(a))$ *pour tout* $a \in \text{dom } \mu'$*.*

Démonstration. La démonstration est extrêmement longue, fastidieuse, ennuyeuse. Nous ne traiterons que quelques-uns des cas saillants de la récurrence structurelle sur la dérivation π.

En particulier, notons que les seuls cas les plus importants sont ceux où π se termine par la règle (Fun), (App) ou $(FixApp)$.

- Dans le cas de (Fun), on doit vérifier que $\alpha_0\langle x, M, \varrho\rangle$ est la fonction qui à (μ'', V) associe $[\![M]\!]_{\text{caml}} (\rho[x \mapsto V])\mu''$ (voir équation (5)). Rappelons que $\rho(y) = \alpha_0(\varrho(y))$ pour tout $y \in \text{fv}(M) \setminus \{x\}$: ce résultat est donc exactement la définition de $\alpha_0\langle x, M, \varrho\rangle$.
- Dans le cas de (App), π se termine par

$$\dfrac{\begin{array}{c}\vdots\ \pi' \\ \varrho, \mu \vdash M_1 \Rightarrow \mu', \langle x, M', \varrho'\rangle\end{array} \quad \begin{array}{c}\vdots\ \pi'' \\ \varrho, \mu' \vdash N \Rightarrow \mu'', V\end{array} \quad \begin{array}{c}\vdots\ \pi''' \\ \varrho'[x \mapsto V], \mu'' \vdash M' \Rightarrow \mu''', V'\end{array}}{\varrho, \mu \vdash M_1 N \Rightarrow \mu''', V'}\ (App)$$

  Par hypothèse de récurrence $[\![M_1]\!]_{\text{caml}} \rho\tilde{\mu}$ est de la forme $(\tilde{\mu}', \phi)$, où : (a) $\text{dom } \mu' \subseteq \text{dom } \tilde{\mu}'$ et $\tilde{\mu}'(a) = \alpha_0(\mu'(a))$ pour tout $a \in \text{dom } \mu'$; et (b) la fonction $\phi = \alpha_0\langle x, M', \varrho'\rangle$ est définie par $\phi(\tilde{\mu}'', \tilde{V}) = [\![M']\!]_{\text{caml}} (\rho''[x \mapsto \tilde{V}])\tilde{\mu}''$, où ρ'' est n'importe quel environnement tel que pour tout $y \in \text{fv}(M') \setminus \{x\}$, $\rho(y) = \alpha_0(\varrho'(y))$.
  Par hypothèse de récurrence encore, $[\![N]\!]_{\text{caml}} \rho\tilde{\mu}'$ est de la forme $(\tilde{\mu}'', \tilde{V})$, où : (c) $\text{dom } \mu'' \subseteq \text{dom } \tilde{\mu}''$ et $\tilde{\mu}''(a) = \alpha_0(\mu''(a))$ pour tout $a \in \text{dom } \mu''$; et (d) $\alpha_0(V) = \tilde{V}$.
  Posons ρ'' n'importe quel environnement tel que pour tout $y \in \text{fv}(M') \setminus \{x\}$, $\rho''(y) = \alpha_0(\varrho'(y))$. L'environnement $\rho''[x \mapsto \tilde{V}]$ est tel que pour tout $y \in (M')$, $\rho''[x \mapsto \tilde{V}](y) = \alpha_0(\varrho'[x \mapsto V](y))$, en utilisant (d). On peut donc appliquer l'hypothèse de récurrence une troisième fois, et conclure que $[\![M']\!]_{\text{caml}} (\rho''[x \mapsto \tilde{V}])\tilde{\mu}''$ est de la forme $(\tilde{\mu}''', \tilde{V}')$, où : (e) $\text{dom } \mu''' \subseteq \text{dom } \tilde{\mu}'''$ et $\tilde{\mu}'''(a) = \alpha_0(\mu'''(a))$ pour tout $a \in \text{dom } \mu'''$; et (f) $\alpha_0(V') = \tilde{V}'$.
  Par définition de ϕ, $(\tilde{\mu}''', \tilde{V}')$ est $\phi(\tilde{\mu}'', \tilde{V}) = \phi([\![N]\!]_{\text{caml}} \rho\tilde{\mu}')$. On conclut par l'équation (4), (e) et (f).
- Le cas où π se termine par $(FixApp)$ est très similaire, et est laissé en exercice au lecteur.
  $\square$