Programmation

Jean Goubault-Larrecq
LSV/CNRS UMR 8643 & INRIA Futurs projet SECSI & ENS Cachan
61 avenue du président-Wilson, F-94235 Cachan Cedex
goubault@lsv.ens-cachan.fr
Phone: +33-1 47 40 75 68 Fax: +33-1 47 40 75 21

21 novembre 2003

Résumé

Ce document sert de notes de cours pour le premier quart du cours de programmation du magistère STIC, ENS Cachan, édition 2003–2004. Il s'agit de la version 1, qui date du 02 octobre 2003 (leçon 1), du 16 octobre 2003 (leçon 2), du 20 octobre 2003 (leçon 3), du 23 octobre 2003 (leçon 4).

Table des matières

1 Leçon 1 **3**
1.1 Une brève introduction aux langages de programmation . . . 3
1.2 Quelques bases de théorie de l'ordre . . . 10
1.2.1 Points fixes et boucles . . . 10
1.2.2 Treillis complets et théorème de Tarski . . . 13
1.2.3 Cpos, fonctions Scott-continues . . . 15

2 Leçon 2 **19**
2.1 Langages impératifs, le langage C . . . 19
2.1.1 Affectations . . . 19
2.1.2 Tableaux, structures . . . 21
2.1.3 Structures de contrôle . . . 24
2.2 Langages fonctionnels, le cas de mini-Caml . . . 27

3 Leçon 3 **32**
3.1 Architecture et assembleur . . . 32
3.1.1 Mémoires . . . 32
3.1.2 Le processeur . . . 35

3.1.3 Modes d'adressage et formats d'instructions 39
3.1.4 Formalisation 42
3.2 La sémantique dénotationnelle (de mini-Caml) 46
3.2.1 Domaines, le modèle $\mathbb{P}\omega$ de Plotkin 47

4 Leçon 4 **52**
4.1 Sémantique dénotationnelle de mini-Caml 52
4.1.1 Quel genre de cpo nous faut-il ? 52
4.1.2 Sémantique dénotationnelle des expressions 55
4.1.3 Sémantique dénotationnelle des programmes 58

5 Leçon 5 **60**
5.1 Sémantique opérationnelle grands pas de mini-Caml 60
5.1.1 Clôtures 60
5.1.2 Appel gauche-droite, par valeur, par nécessité, par référence 63
5.1.3 Autres règles 64
5.1.4 Sémantique opérationnelle des programmes 65
5.1.5 Arbres de dérivations, récurrence sur les dérivations 66
5.1.6 Règles dérivées 69
5.1.7 Correction de la sémantique concrète par rapport à l'abstraite 72

A Guide de référence rapide de l'assembleur Pentium **76**

1 Leç on 1

1.1 Une brève introduction aux langages de programmation

Il existe de nombreux langages de programmation. En fait, en comptant les langages d'usage général, les langages de script, les langages dédiés à une application, ..., le nombre de langages existants se compte en dizaines de milliers.

Aujourd'hui, les langages les plus couramment répandus sont C, C++ et Java. Il s'agit de langages *impératifs*, c'est-à-dire où l'exécution procède par changements d'états successifs. Par exemple, la factorielle (un exemple que je reprendrai jusqu'à la nausée dans la suite de ce cours) s'écrit en C :

```
int fact (int n)
{
  int resultat;
  int i;

  resultat = 1;
  for (i=1; i<=n; i++)
    resultat = resultat * i;
  return resultat;
}
```

ce qui se lit informellement : "pour calculer la factorielle de n, mettre le résultat à 1, puis le multiplier successivement par 1, par 2, ..., par n".

À l'opposé, les langages *déclaratifs* tentent de décrire ce qu'on souhaite calculer sans décrire comment — ou, sans être aussi extrémiste, insistent sur le quoi au détriment du comment. Par exemple, la factorielle en un langage *fonctionnel* comme CaML s'écrira :

```
let rec fact n =
        if n=0
           then 1
        else n * fact (n-1);;
```

ce qui correspond bien à la notation mathématique définissant la factorielle :

$$n! = \begin{cases} 0 & \text{si } n = 0 \\ n.(n-1)! & \text{si } n \neq 0 \end{cases}$$

Un autre exemple est donné par les langages *logiques*, dont le représentant typique est Prolog, où l'on écrirait typiquement :

```
fact(0,1).
fact(N+1,Y) :- fact(N,Z), Y=(N+1)*Z.
```

(J'ai en réalité un peu triché dans l'écriture de ce programme, pour des raisons de lisibilité.) Ceci définit une relation `fact` entre deux entiers, telle que $\texttt{fact}(m, n)$ est censé être vrai si $n = m!$, et définie par le fait que $1 = 0!$, et $Y = (N + 1)!$ s'il existe $Z = N!$ tel que $Y = (N + 1)Z$. Je n'insisterai pas sur les langages logiques, que vous comprendrez mieux de toute faç on après avoir suivi le cours de logique d'Hubert Comon.

Le but de ce cours n'est pas spécifiquement de vous apprendre à programmer dans tel ou tel langage de programmation, mais de vous apprendre quelques concepts de base communs à pratiquement tous les langages de programmation.

Un concept central est celui de *sémantique*. Plutôt que de dire de quoi il s'agit tout de suite, examinons les questions suivantes :

– Je vous donne un programme, par exemple le programme `fact` écrit en C ou en CaML comme ci-dessus. Que fait ce programme ?
  Vous pouvez vous laisser guider par une intuition plus ou moins vague de ce que fait chaque instruction du programme. Mais ceci ne mène pas toujours loin. Par exemple, sachant que, en C, x++ ajoute un au contenu de la variable x et retourne la valeur qu'avait x avant, que fait l'instruction suivante ?
  `x = x++;`
  Vous pouvez tester ce programme, et par exemple sur ma machine, ceci ajoute juste 1 à `x`. Il est probable que, si vous avez un compilateur C qui fonctionne comme le mien, vous soyez convaincus que c'est effectivement ce que doit faire cette instruction. (J'expliquerai la notion de compilateur un peu plus loin.) Mais un jour vous le ferez peut-être tourner sur une autre machine, ou avec un autre compilateur, et l'instruction ci-dessus ne changera pas la valeur de `x`. La raison en est que `x = x++;` demande essentiellement à effectuer trois opérations :

  1. lire la valeur de `x` avant, appelons-la x_0 ;
  2. réinscrire $x_0 + 1$ dans la variable `x` (l'effet de `x++`) ;
  3. mettre x_0 (la valeur de `x++`) dans `x`.

  Or les deux dernières entrent en conflit : on n'obtient pas le même résultat selon qu'on exécute 1 avant 2 ou 2 avant 1, et la norme du langage C ne précise aucun ordre entre ces actions. Plus surprenant, il n'est pas exclu qu'un compilateur décide de traduire cette instruction sous forme d'instructions assembleur qui mettent essentiellement *n'importe quoi* dans `x`.
  Savoir exactement les effets possibles d'une instruction donnée, c'est en connaître la sémantique.

– J'ai écrit deux versions de la factorielle, en C et en CaML, ci-dessus. Comment puis-je être sûr qu'ils calculent la même chose ?
  Évidemment, j'ai tout fait pour, donc ils devraient calculer la même chose... mais j'ai pu me tromper (il peut y avoir des *bogues*). Comment sais-je que je ne me suis pas trompé, ou comment puis-je trouver les bogues ? Il y a plusieurs approches : utiliser un *débogueur*, ou effectuer des campagnes de tests, permettent de détecter ou de comprendre des erreurs, du moins si ces erreurs se manifestent suffisamment fréquemment et de faç on suffisamment reproductible.

On peut aussi tenter de démontrer un théorème d'équivalence des deux programmes. Ceci revient à démontrer que les deux programmes ont la *même sémantique*, ou dans les cas plus complexes, ont des sémantiques reliées par des relations ayant de bonnes propriétés (relations d'abstraction, de raffinement, bisimulations ; ce ne sera pas le programme de ce cours).

Pour ceci, on aura besoin de définir *mathématiquement* les diverses sémantiques de nos langages de programmation.

– J'ai parlé un peu plus haut de compilateurs (vers le langage assembleur). Un compilateur est un programme, qui prend le texte source d'un programme en entrée (par exemple en C ou en CaML), et le traduit en un autre langage (typiquement le langage machine, ou assembleur). Une sémantique, c'est fondamentalement pareil : c'est une fonction prenant en entrée un texte source et retournant un objet mathématique.

  Si l'on arrive à considérer les programmes (C, CaML, assembleur) comme des objets mathématiques, les compilateurs ne seront rien d'autre que des fonctions de sémantique particulières. L'avantage de cette vision sera en particulier que l'on pourra démontrer qu'un compilateur est correct par rapport à une sémantique : ceci reviendra à montrer que deux sémantiques sont équivalentes (typiquement, que ce sont deux fonctions égales).

Ce cours abordera donc des extrêmes assez éloignés :

– D'un côté nous aurons un peu de mathématiques : sémantiques dénotationnelles, opérationnelles ; théorie de l'ordre et théorèmes de points fixes, théorie de la démonstration et récurrences structurelles.
– De l'autre nous aurons à voir quelques considérations très pratiques : pour comprendre la sémantique de l'assembleur, il faudra comprendre comment fonctionne un ordinateur : processeur, mémoire, bus, portes logiques et flip-flops. C'est l'*architecture* des ordinateurs, à laquelle Béatrice Bérard vous donnera une introduction en deux séances de quatre heures.

Avant de passer à la suite, considérez le programme suivant en C, si vous comprenez déjà le C (on décortiquera ce programme en cours) :

```
void merge (int *l1, int n1,
            int *l2, int n2,
            int *res) {
  while (n1!=0 && n2!=0) {
    if (*l1 < *l2)
       { *res++ = *l1++; n1--; }
    else { *res++ = *l2++; n2--; }
  }
  while (n1--!=0) *res++ = *l1++;
  while (n2--!=0) *res++ = *l2++;
}
```

```
void sort1 (int *l, int n,
            int *res) {
  int k;

  if (n==0) return;
  if (n==1) { *res = *l; return; }
  k = n / 2;
  sort1 (l, k, res);
  sort1 (l+k, n-k, res+k);
  merge (res, k, res+k, n-k, l);
}
```

```
void sort (int *l, int n) {
  int *aux = (int *) malloc (n * sizeof (int));
  sort1 (l, n, aux);
  free (aux);
}
```

Il s'agit d'un programme de tri d'un tableau `l` d'entiers, de longueur `n`, par la méthode dite de *tri par fusion*. Ce programme contient un gros bogue : lequel ? Vous pouvez le trouver par test et déboguage, ou en raisonnant formellement. Dans tous les cas, je ne pense pas que vous voyiez lequel il est en moins de cinq minutes. Ceci nous mène à l'idée qu'on aimerait disposer de méthodes automatiques de preuve de programme. Ce sera en particulier le sujet du cours d'analyse statique, une famille de techniques permettant de détecter certaines propriétés simples d'un programme, et celui des logiques de Hoare et variantes, qui seront vues plus tard dans ce cours de programmation.

Un autre exemple que je prendrai dans ce cours est le petit programme de la figure 1.

Tapez-le dans un fichier que vous nommerez `cat.c`. Il s'agit d'un autre exemple que je réutiliserai jusqu'à la nausée. Il s'agit du texte source de l'utilitaire `cat` d'Unix. (Du moins, d'une version naïve, mais qui fonctionne.) Ce programme est censé s'utiliser en tapant sous le *shell* (interprète de commandes Unix), par exemple :

```
cat a b
```

ce qui va afficher le contenu du fichier de nom `a`, suivi du contenu du fichier de nom `b`. (Vous aurez pris soin de créer deux fichiers nommés `a` et `b` d'abord, bien sûr.) En général, `cat` suivi de noms de fichiers $f_1, \ldots, f_n$, va afficher les contenus des fichiers $f_1, \ldots, f_n$ dans l'ordre ; ceci les `cat`ène.

Si, au lieu d'utiliser l'utilitaire Unix `cat`, vous essayez d'utiliser le programme `cat.c` ci-dessus, par exemple en tapant :

```
cat.c a b
```

vous verrez que cela ne fonctionne pas. C'est parce que le *processeur* de la machine, c'est-à-dire le circuit intégré qui fait tous les calculs dans la machine en face de vous (Pentium, PowerPC, ou

```c
#include <stdio.h>

main (int argc, char *argv[])
{
  int i, c;

  for (i=1; i<argc; i++)
    {
      FILE *f;

      f = fopen (argv[i], "r");
      while ((c = fgetc (f))!=EOF)
        fputc (c, stdout);
      fclose (f);
    }
  fflush (stdout);
  exit (0);
}
```

FIG. 1 – Un programme de concaténation en C

autre), ne comprend pas le texte, compréhensible aux humains (du moins on l'espère) du fichier `cat.c`.

Pour savoir ce que le processeur comprend, on peut aller voir à quoi ressemble l'utilitaire `cat`, le vrai. D'abord, on cherche où il se trouve (c'est un fichier comme un autre !), en demandant `which cat`. Sur ma machine, la réponse est `/bin/cat`. Si je charge `/bin/cat` sous XEmacs, un éditeur de texte, j'obtiens quelque chose qui ressemble à :

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@
‘\211^D^H4^@^@^@x!^@^@^@^@^@^@4^@ ^@^F^@(^@^X
^@^W^@^F^@^@^@4^@^@^@4\200^D^H4\200^D^HÀ^@^@^@
À^@^@^@^E^@^@^@^D^@^@^@^C^@^@^@ô^@^@^@ô\200^D
^Hô\200^D^H^S^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A
^@^@^@^@^@^@^@^@\200^D^H^@\200^D^H3^_^@^@3^_^@
^@^E^@^@^@^@^P^@^@^A^@^@^@4^_^@^@4¯^D^H4¯^D^Ht
^A^@^@,^B^@^@^F^@^@^@^@^P^@^@^B^@^@^@^H ^@^@^H
°^D^H^H°^D^H ^@^@^@ ^@^@^@^F^@^@^@^D^@^@^@^D^@
^@^@^H^A^@^@^H\201^D^H^H\201^D^H ^@^@^@ ^@^@^@
^D^@^@^@^D^@^@^@/lib/ld-linux.so.2^@^@^D^@^@^@
^P^@^@^@^A^@^@^@GNU^@^@^@^@^@^B^@^@^@^@^@^@^@
```

etc. Peu importe de ce que cela signifie exactement pour le moment, le point important est que, si ceci est compréhensible pour le processeur, c'est loin d'être lisible pour un être humain ! A

contrario, le programme `cat.c` est bien plus lisible, mais le processeur ne le comprend pas.

La traduction du fichier `cat.c` (le *source*) en `cat` (l'*exécutable*) se fait au moyen d'un *compilateur*. Par exemple, la commande `gcc` est le compilateur C, et si l'on tape :

```
gcc -o mycat cat.c
```

le compilateur `gcc` va traduire (compiler) le source `cat.c` en un exécutable que j'ai décidé de nommer `mycat`, pour ne pas prêter à confusion avec l'utilitaire standard `cat`. Vous pouvez alors lancer `mycat`, et vérifier qu'il donne bien les résultats attendus en tapant la ligne de commande suivante, et en comparant avec ce qui se passait avec le programme `cat` :

```
./mycat a b
```

(Le "`./`" avant `mycat` nous sert à dire au shell que l'outil de nom `mycat` à utiliser est celui qui est dans notre répertoire courant.)

▸ **EXERCISE 1.1**

Que se passe-t-il si vous tapez la ligne suivante ?

```
./mycat cat.c
```

Vous pouvez aussi voir comment fonctionne `mycat` en utilisant le débogueur `ddd` :

▸ **EXERCISE 1.2**

Sous Unix, avec `ddd` installé (sinon, utilisez `gdb`, mais c'est moins lisible. . .) :

- Recompilez `mycat` en tapant `gcc -ggdb -o mycat cat.c`. L'option `-ggdb` permettra au débogueur de vous afficher des informations pertinentes (pour plus d'information, tapez `man gcc`).
- Tapez `ddd mycat &`. La fenêtre de `ddd` s'ouvre, montrant le texte source `cat.c`. Fermez la fenêtre "Tip of the Day" si besoin est.
- Cliquez sur le début de la ligne `int i, c;` avec le bouton droit de la souris, déroulez le menu qui s'affiche, et cliquez sur "set breakpoint". Une petite icône figurant un panneau stop rouge s'affiche par-dessus la ligne.
- En plaç ant la souris sur la case du bas, où s'est affiché le message :
  ```
  (gdb) break cat.c:5
  Breakpoint 1 at 0x8048536: file cat.c, line 5.
  (gdb)
  ```
  tapez `run a b`. Ceci lance le programme `mycat`, avec arguments les chaînes de caractères `a` et `b`. Le programme s'arrête sitôt lancé sur le "breakpoint" que l'on a mis ci-dessus.
- Tirez le menu "View", et cliquez sur "Data Window" ; cliquez ensuite sur les menus "Data/Display Arguments" et "Data/Display Local Variables". Pour voir les arguments ("`a`", "`b`") qui ont été passés au programme `mycat`, cliquez sur le menu "Data/Memory...", et écrivez 3 en face de "Examine", puis "string" au lieu de "octal", enfin tapez `argv[0]` dans la case après "from", puis cliquez sur les boutons "Display" puis "Close". Vous aurez probablement besoin de réorganiser le cadre du haut de la fenêtre de `ddd` pour bien voir les affichages "Args", "Locals" et "X".

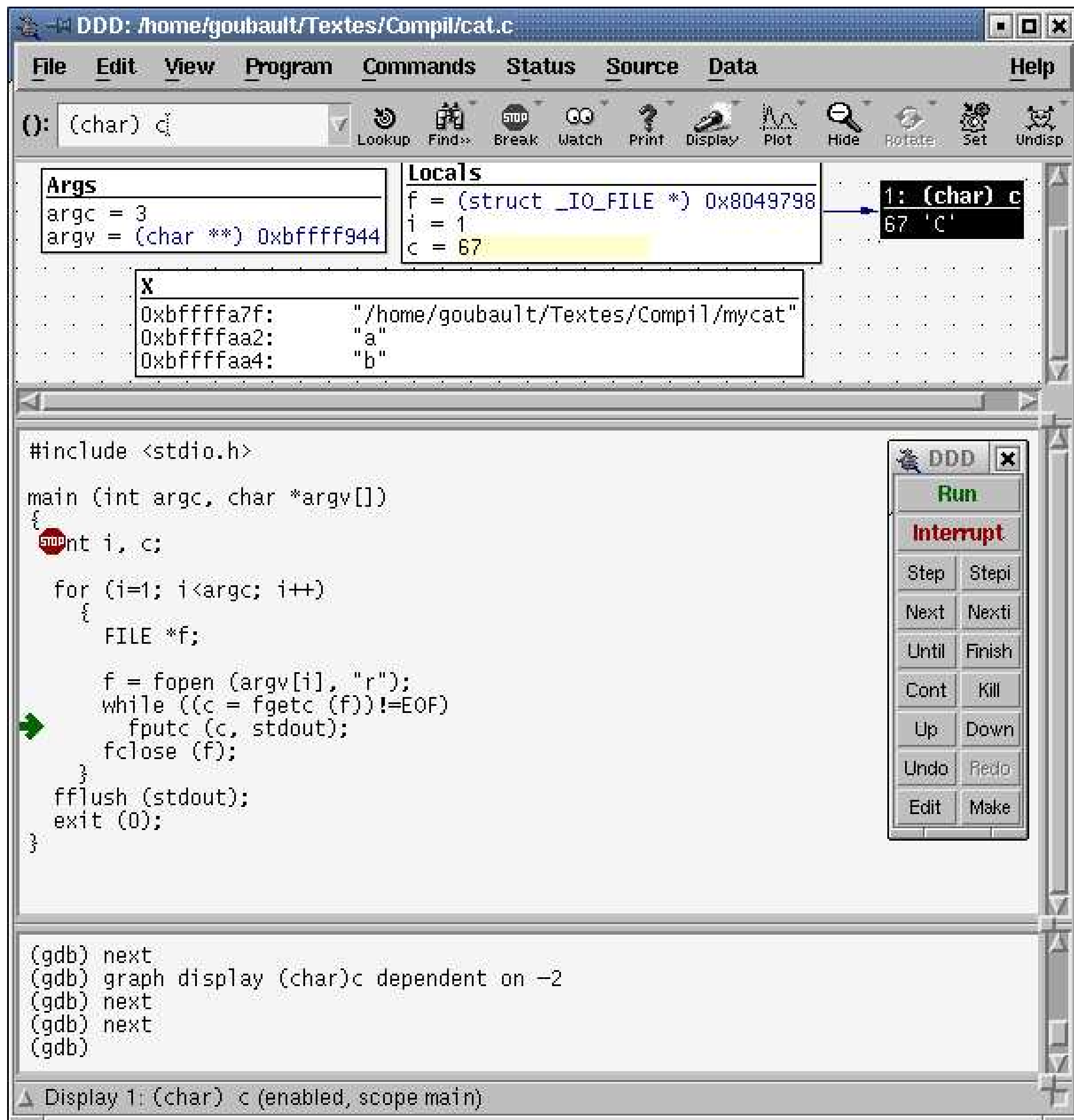


FIG. 2 – Une session sous ddd

– Cliquez ensuite sur le bouton “Next” de la fenêtre flottante “DDD” répétitivement pour voir le programme s'exécuter petit à petit. Si vous voulez voir le contenu de la variable `c` sous forme d'un caractère lisible plutôt que sous forme d'un entier, cliquez avec le bouton droit sur la ligne de `c` dans l'affichage “Locals”, tirez le menu local “Display/Others...”, écrivez `(char)c` dans le cadre sous “Display Expression”, puis cliquez sur “Display”.

Vous devriez obtenir quelque chose qui ressemble à la figure 2 ; Vous pouvez bien sûr effectuer la même manipulation avec d'autres programmes, y compris ceux de factorielle ou de tri décrits plus hauts.

1.2 Quelques bases de théorie de l'ordre

Pour vous reposer des aspects pratiques de la section précédente, on va faire ici un peu de mathématiques... mathématiques qui nous serviront dans la suite.

1.2.1 Points fixes et boucles

Sans dire tout de suite ce qu'est une sémantique, imaginons qu'il s'agit d'une fonction $[\![_]\!]$ qui à chaque programme P associe une valeur $[\![P]\!]$ dans un certain domaine de *valeurs* D. Un des intérêts de cette approche, qui s'appelle la *sémantique dénotationnelle* et qui date de la fin des années 1960, c'est qu'en principe on pourra écrire et prouver des théorèmes de la forme :

$$[\![P_1]\!] = [\![P_2]\!] \qquad \text{(“les programmes } P_1 \text{ et } P_2 \text{ font la même chose”)}$$

ou bien

$$[\![P_1]\!] \neq [\![P_2]\!] \qquad \text{(“les programmes } P_1 \text{ et } P_2 \text{ ne font pas la même chose”)}$$

par exemple.

L'un des problèmes centraux à résoudre dans une telle approche sera de donner une sémantique aux *boucles*. Considérons en effet la boucle `while` du langage C. Cette construction prend la forme

```
while (b) e
```

où `b` est une expression booléenne (retournant vrai ou faux), et `e` est un bloc d'instructions écrites en C. Une telle boucle fait les opérations suivantes :

1. Calculer `b`.
2. Si le résultat vaut vrai, alors exécuter les instructions de `e`, et revenir à l'étape 1.
3. Sinon, sortir de la boucle.

Par exemple, la boucle

```
while (n1!=0 && n2!=0) {
  if (*l1 < *l2)
    { *res++ = *l1++; n1--; }
  else { *res++ = *l2++; n2--; }
  }
```

de la fonction merge teste si n1 et n2 sont tous les deux non nuls ; si c'est le cas, les trois lignes suivantes sont exécutées, puis l'on revient au début de la boucle ; sinon, la boucle est terminée. Un autre exemple est donné par la boucle

```
while ((c = fgetc (f))!=EOF)
  fputc (c, stdout);
```

du programme cat.c. La condition booléenne b est (c = fgetc (f))!=EOF, qui lit le caractère suivant dans le fichier f par fgetc (f), le stocke dans la variable c (c = fgetc (f)), puis retourne vrai si et seulement si ce caractère est différent du marqueur de fin de fichier EOF. Si ce test réussit, autrement dit si l'on n'est pas encore à la fin du fichier f, l'instruction e, à savoir fputc (c, stdout);, est exécutée : elle affiche le caractère c sur le fichier stdout (la *sortie standard*), c'est-à-dire (en général) sur votre écran.

Maintenant, il semble intuitif que while (b) e devrait faire la même chose que le programme

```
if (b)
  { e; while (b) e }
```

qui teste d'abord si b est vrai, et si c'est le cas exécute e puis la boucle proprement dite. (On dit qu'on a *déroulé la boucle* un coup.) Ceci se manifeste par le fait que l'on souhaiterait que l'équation suivante soit vraie :

$$[\![\texttt{while}\,(\texttt{b})\,\texttt{e}]\!] = [\![\texttt{if}\,(\texttt{b})\,\{\texttt{e};\ \texttt{while}\,(\texttt{b})\,\texttt{e}\ \}]\!]$$

Notons x l'inconnue $[\![\texttt{while}\,(\texttt{b})\,\texttt{e}]\!]$. Il est intuitivement naturel de penser que ceci signifie que x et if (b) { e; x} devraient avoir la même sémantique. Si l'on note F la fonction qui à tout programme x associe if (b) { e; x}, on voit que nous cherchons x sous la forme d'un *point fixe* de F : x et $F(x)$ doivent avoir la même sémantique.

Cet argument n'est pas très rigoureux. Plus formellement, étendons la syntaxe des programmes de sorte qu'ils puissent contenir des *paramètres* X_1, ..., X_k dénotant des programmes non encore explicités. Appelons *contexte* tout k-uplet $(v_1, \ldots, v_k)$ de valeurs (éléments de D). On peut imaginer étendre la fonction de sémantique $[\![_]\!]$ à une fonction prenant un programme P à k paramètres, et un contexte $(v_1, \ldots, v_k)$, et retournant une valeur $[\![P]\!]\,(v_1, \ldots, v_k)$, dénotant intuitivement la valeur du programme P lorsque X_1 vaut v_1, ..., X_k vaut v_k. (Cette intuition se formalise par l'équation $[\![P]\!]\,([\![Q_1]\!], \ldots, [\![Q_k]\!]) = [\![P[X_1 \leftarrow Q_1, \ldots, X_k \leftarrow Q_k]]\!]$, où $\leftarrow$ dénote la substitution.) Supposons $k = 1$ pour simplifier. Soit F la fonction qui à $v \in D$ associe $[\![\texttt{if}\,(\texttt{b})\,\{\texttt{e};\,X_1\ \}]\!]\,(v)$. Le déroulement de boucle revient à demander que $x = F(x)$, ce qui est l'énoncé précis du fait que x est un point fixe.

Un autre exemple où l'on aura besoin de la notion de point fixe est dans la définition de fonctions récursives. Considérons le calcul de factorielle 7 en Caml :

```
let rec fact n =
        if n=0
           then 1
        else n * fact (n-1)
in fact 7;;
```

On voit que `fact` est une fonction qui est définie en termes d'elle-même : la définition de `fact`, après le premier signe =, fait appel à `fact` (que l'on applique à `n-1`). (Le fait que `fact` soit définie en termes d'elle-même est la raison d'être du mot-clé `rec`, qui instruit le compilateur Caml de ce fait.) C'est peut-être un peu plus clair si l'on réécrit l'expression Caml ci-dessus en :

```
let rec fact =
        fun n -> if n=0
                    then 1
                 else n * fact (n-1)
in fact 7;;
```

où la construction `fun n ->` [...] dénote la fonction qui à `n` associe [...]. Remplaç ons `fact` par l'inconnue x dans sa définition, et soit F la fonction qui au programme $\underline{x}$ associe le programme

```
fun n -> if n=0
            then 1
         else n * x (n-1)
```

On a donc défini `fact` comme étant un point fixe de F.

Les deux exemples des boucles `while` et des fonctions récursives ne sont pas si éloignés qu'il y paraît. La boucle `while (b) e` peut en effet se coder en CaML sous la forme :

```
let rec boucle () =
        if (b)
           then (e; boucle ())
        else ()
in
   boucle ();;
```

Rappelez-vous comment on calcule le point fixe d'une fonction contractante F de $\mathbb{R}$ dans $\mathbb{R}$: partant de n'importe quel point x_0, on calcule les itérés $F^n(x_0)$ (où $F^0(x) = x$, $F^{n+1}(x) = F(F^n(x))$), et la limite est l'unique point fixe de F. Nous ne pourrons pas supposer que nous sommes dans un espace métrique complet dans la suite, ni que la fonction F qui à un programme x associe `if (b) { e;` x`}` est contractante, mais nous tenterons de calculer la sémantique de la boucle comme une sorte de limite des itérés $F^n(x_0)$ pour une certaine valeur x_0. Observons que cela revient à dire que la sémantique de `while (b) e` est la "limite" de

$$
\begin{gathered}
x_0 \\
\texttt{if}\ (\texttt{b})\ \{\ \texttt{e};\ x_0\} \\
\texttt{if}\ (\texttt{b})\ \{\ \texttt{e};\ \texttt{if}\ (\texttt{b})\ \{\ \texttt{e};\ x_0\}\} \\
\texttt{if}\ (\texttt{b})\ \{\ \texttt{e};\ \texttt{if}\ (\texttt{b})\ \{\ \texttt{e};\ \texttt{if}\ (\texttt{b})\ \{\ \texttt{e};\ x_0\}\}\} \\
\dots
\end{gathered}
$$

Le problème de base va être de trouver des catégories de domaines de valeurs où toute fonction (raisonnable) aura au moins un point fixe.

1.2.2 Treillis complets et théorème de Tarski

L'une des catégories les plus simples où l'on aura un théorème du point fixe est celle des treillis complets et des fonctions monotones.

Rappelons qu'une relation d'*ordre* $\leq$ sur un ensemble X est une relation binaire réflexive ($x \leq x$), antisymétrique ($x \leq y$ et $y \leq x$ impliquent $x = y$), et transitive ($x \leq y$ et $y \leq z$ impliquent $x \leq z$). Un *ensemble ordonné* est un couple $(X, \leq)$ formé d'un ensemble X et d'une relation d'ordre $\leq$ sur X. On notera souvent X au lieu de $(X, \leq)$, par abus de langage. Une fonction $f : X \to Y$ sera dite *monotone* si et seulement si elle préserve l'ordre : si $x \leq x'$, alors $f(x) \leq f(x')$.

Rappelons aussi qu'un *majorant* y d'une partie F de X est un élément supérieur ou égal à tout élément de X : $x \leq y$ pour tout $x \in X$. Un *minorant* y est inférieur ou égal à tout élément de X : $y \leq x$ pour tout $x \in X$. La borne supérieure de F est le plus petit des majorants de F dans X, si elle existe ; elle est alors nécessairement unique. De même, la borne inférieure de F est le plus grand des minorants de F dans X, si elle existe ; elle est alors nécessairement unique.

Définition 1 (Treillis complet) *Un* treillis complet *est un ensemble ordonné* $(X, \leq)$ *non vide tel que toute famille* $F \subseteq X$ *a une borne supérieure* $\bigvee F$ *et une borne inférieure* $\bigwedge F$.

Si $F = \emptyset$, $\bigwedge \emptyset$ est par définition le plus grand élément de X, et sera noté $\top$; $\bigvee \emptyset$ est le plus petit élément de X, et sera noté $\bot$. On notera aussi $\bigvee_{i \in I} x_i$ la borne supérieure de la famille $(x_i)_{i \in I}$, et de même $\bigwedge_{i \in I} x_i$ sa borne inférieure. On notera aussi $x \vee y = \bigvee \{x, y\}$ et $x \wedge y = \bigwedge \{x, y\}$. Pour faire court, on dira aussi "le sup" au lieu de "la borne supérieure", et "l'inf" pour "la borne inférieure".

▸ **EXERCISE 1.3**

Un *inf-demi-treillis complet* est un ensemble ordonné $(X, \leq)$ non vide tel que toute famille $F \subseteq X$ a une borne inférieure $\bigwedge F$. Montrer que toute famille $F \subseteq X$ a une borne supérieure $\bigvee F$, et que donc tout inf-demi-treillis complet est un treillis complet. (Indication : $\bigvee F$ s'il existe est le plus petit des majorants... comment peut-on donc le construire ?)

▸ **EXERCISE 1.4**

Montrer que tout sup-demi-treillis complet (notion que vous définirez par analogie avec la précédente) est un treillis complet.

▸ **EXERCISE 1.5**

Soit A un ensemble quelconque, $\mathbb{P}(A)$ l'ensemble de ses parties. Montrer que $(\mathbb{P}(A), \subseteq)$ est un treillis complet.

▸ **EXERCISE 1.6**

Par l'exercice 1.4, l'ensemble des ouverts $\mathcal{O}$ d'un espace topologique $(X, \mathcal{O})$ est un treillis complet : le sup d'une famille F d'ouverts est juste leur union. Quel est son inf ?

Le point important est que toute fonction monotone d'un treillis complet dans lui-même a un point fixe :

Théorème 1 (Tarski-Knaster) *Soit $(X, \leq)$ un treillis complet. Soit $f : X \to X$ une fonction monotone. Alors f a un plus petit point fixe $lfp(f)$ et un plus grand point fixe $gfp(f)$. De plus l'ensemble $Fix(f)$ de tous les points fixes de f, ordonnés par $\leq$, est treillis complet.*

Avant d'en faire la démonstration, considérons l'intuition donnée à la fin de la section 1.2.1. Partant de $\bot$, on va calculer $f(\bot)$, $f^2(\bot)$, ..., $f^n(\bot)$, ... Le sup de cette famille, $\bigvee_{n\in\mathbb{N}} f^n(\bot)$, vérifie :

$$\begin{aligned} f(\bigvee_{n\in\mathbb{N}} f^n(\bot)) &\geq f(f^n(\bot)) \qquad \text{(pour tout } n\text{)} \\ &= f^{n+1}(\bot) \end{aligned}$$

donc $f(\bigvee_{n\in\mathbb{N}} f^n(\bot)) \geq \bigvee_{n\in\mathbb{N}} f^n(\bot)$, mais l'inégalité inverse ne tient pas en général (voir exercice 1.10). On pourrait utiliser cette forme d'argument en itérant f de faç on transfinie, mais ceci nécessiterait que nous parlions d'ordinaux... et ce n'est pas un cours de théorie des ensembles.

Démonstration. Considérons l'ensemble $Post(f) = \{x \in X | f(x) \leq x\}$ des *post-points fixes* de f. D'abord remarquons que $Post(f)$ est non vide, car $\top$ est dans $Post(f)$; ceci n'a pas en réalité beaucoup d'importance.

Puisque X est un treillis complet, $Post(f)$ a une borne inférieure ; appelons-la x_0. Comme x_0 est un minorant de $Post(f)$, $x_0 \leq x$ pour tout $x \in Post(f)$. Comme f est monotone, $f(x_0) \leq f(x)$, et comme $x \in Post(f)$, $f(x) \leq x$: donc $f(x_0) \leq x$. Ceci étant vrai pour tout $x \in Post(f)$, $f(x_0)$ est un minorant de $Post(f)$. Comme x_0 est le plus grand de tous ces minorants, $f(x_0) \leq x_0$. Mais ceci, par définition, signifie que x_0 est dans $Post(f)$, et est donc le *plus petit post-point fixe* de f.

Revenons sur le fait que $f(x_0) \leq x_0$. Par monotonie de f encore, $f(f(x_0)) \leq f(x_0)$, donc $f(x_0)$ est aussi un post-point fixe de f. Comme x_0 est le plus petit, $x_0 \leq f(x_0)$. Donc $x_0 = f(x_0)$ par antisymétrie. Ceci montre que x_0 est un point fixe de f.

C'est nécessairement le plus petit de tous les points fixes : tout autre point fixe x de f est clairement un post-point fixe de f, et est donc supérieur ou égal au plus petit post-point fixe x_0.

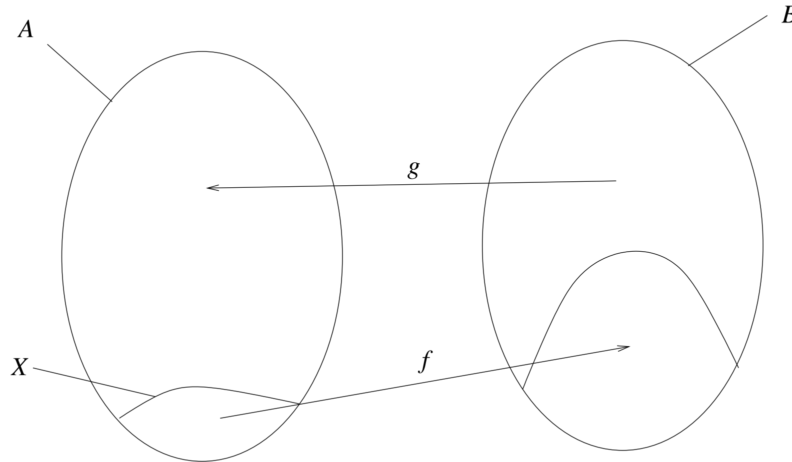
La fin de la démonstration est laissée en exercice. □

▸ EXERCISE 1.7

Terminez la démonstration du théorème de Knaster-Tarski. (Attention : le sup dans X d'une famille de points fixes de f n'est pas nécessairement un point fixe de f, et il faudra donc construire le sup *dans* $Fix(f)$ d'une famille de points fixes de f légèrement différemment ; indication : pensez post-points fixes...)

▸ EXERCISE 1.8

Démontrez le théorème de Cantor-Schröder-Bernstein en utilisant le théorème de Knaster-Tarski : si f est une injection de A dans B et g une injection de B dans A, alors il existe une bijection entre A et B. Indication : trouvez une partie X de A vérifiant certaines propriétés suggérées par la figure ci-dessous, en vous souvenant que $(\mathbb{P}(A), \subseteq)$ est un treillis complet (exercice 1.5).



▸ **EXERCISE 1.9**
Soit $f : Y \times X \to X$ une fonction monotone, au sens où $y \leq y'$ et $x \leq x'$ impliquent $f(y, x) \leq f(y', x')$. Par le théorème de Knaster-Tarski, $lfp(f(y, _))$ existe pour tout y, où $f(y, _)$ dénote la fonction monotone qui à x associe $f(y, x)$. Montrer que la fonction qui à y associe $lfp(f(y, _))$ est monotone de Y dans X. (Indication : utiliser la caractérisation du plus petit point fixe comme plus petit post-point fixe.)

▸ **EXERCISE 1.10**
Soit X l'intervalle $[0, 1]$ de la droite réelle, muni de son ordre naturel $\leq$. Montrer que X est un treillis complet. Si f est une fonction monotone de X dans X, montrer que $f(\bigvee_{n\in\mathbb{N}} x_n) = \bigvee_{n\in\mathbb{N}} f(x_n)$ pour toute suite croissante $(x_n)_{n\in\mathbb{N}}$ si et seulement si f est continue à gauche sur $[0, 1]$. En choisissant judicieusement f, en déduire que $f(\bigvee_{n\in\mathbb{N}} f^n(\bot))$ est en général différent de $\bigvee_{n\in\mathbb{N}} f^n(\bot)$.

1.2.3 Cpos, fonctions Scott-continues

Historiquement, Dana Scott avait proposé d'utiliser les treillis complets comme domaines de valeurs D servant à donner des sémantiques aux langages de programmation. Une notion importante découverte par D. Scott est que toutes les fonctions calculables de D dans D sont non seulement monotones mais encore *continues* (au sens de Scott, voir plus loin). Plus tard, d'autres, et notamment Gordon Plotkin, se sont aperç us que l'on pouvait se passer de treillis complets, et utiliser des ensembles ordonnés plus généraux : les *cpos* ("complete partial order").

Définition 2 (Famille dirigée, fonctions Scott-continues) *Soit $(X, \leq)$ un ensemble ordonné. Une famille $D \subseteq X$ est dite* dirigée *si et seulement si :*
- *D est non vide ;*
- *et pour tous x, y dans D, x et y ont un majorant dans D.*

Une fonction $f : X \to Y$ est Scott-continue *si et seulement si, pour toute famille dirigée D qui a un sup $\bigvee D$, la famille $f(D)$ a un sup et $f(\bigvee D) = \bigvee f(D)$.*

Cette définition est une généralisation, qui s'est avérée naturelle, de ce que nous voulons vraiment représenter. Reprenons la construction du plus petit point fixe sous forme de $\bigvee_{n\in\mathbb{N}} f^n(\bot)$, construction qui avait échoué en section 1.2.2. La famille $(f^n(\bot))_{n\in\mathbb{N}}$ est dirigée, d'abord : c'est facile, et c'est le contenu de l'exercice 1.11 ci-dessous. Si cette famille a un sup $\bigvee_{n\in\mathbb{N}} f^n(\bot)$, la Scott-continuité de f est exactement l'hypothèse supplémentaire dont nous avions besoin pour montrer que $\bigvee_{n\in\mathbb{N}} f^n(\bot)$ est le plus petit point fixe de f : c'est le lemme 2.

▸ **EXERCISE 1.11**
Soit $(x_n)_{n\in\mathbb{N}}$ une suite croissante quelconque dans $(X, \leq)$. Autrement dit, $x_n \leq x_{n+1}$ pour tout $n \in \mathbb{N}$. Montrer que $(x_n)_{n\in\mathbb{N}}$ forme une famille dirigée. Si $f : X \to X$ est une fonction monotone, montrer que $(f^n(\bot))_{n\in\mathbb{N}}$ est une suite croissante, et forme donc une famille dirigée.

Lemme 2 *Soit f une fonction Scott-continue de X dans X, et supposons que X a un plus petit élément $\bot$. Si $\bigvee_{n\in\mathbb{N}} f^n(\bot)$ existe, c'est le plus petit point fixe de f.*

Démonstration. C'est un point fixe :

$$\begin{aligned} f(\bigvee_{n\in\mathbb{N}} f^n(\bot)) &= \bigvee_{n\in\mathbb{N}} f(f^n(\bot)) && \text{(par Scott-continuité)} \\ &= \bigvee_{n\in\mathbb{N}} f^{n+1}(\bot) = \bigvee_{n\in\mathbb{N}} f^{n+1}(\bot) \vee \bot && \text{(trivialement)} \\ &= \bigvee_{n\in\mathbb{N}} f^n(\bot) \end{aligned}$$

C'est le plus petit : supposons en effet que x est un point fixe, et commenç ons par montrer que $f^n(\bot) \leq f^n(x)$ pour tout n, par récurrence sur n ; c'est évident si $n = 0$, et par la monotonie de f dans le cas récurrent. On en déduit que $\bigvee_{n\in\mathbb{N}} f^n(\bot) \leq \bigvee_{n\in\mathbb{N}} f^n(x)$, or $\bigvee_{n\in\mathbb{N}} f^n(x) = \bigvee_{n\in\mathbb{N}} x = x$ puisque $f(x) = x$. Donc $\bigvee_{n\in\mathbb{N}} f^n(\bot) \leq x$. □

L'intuition derrière la notion de Scott-continuité est qu'un programme *calcule* une valeur, et qu'en particulier une boucle devrait *calculer* sa valeur en tant que point fixe. La construction du théorème de Knaster-Tarski est non constructive. L'intérêt de la construction du lemme 2 est que, en identifiant sup et limite, le plus petit point fixe est construit comme une limite d'itérés finis $f^n(\bot)$ de la fonction f, ce qui correspond à l'intuition que nous avons donnée à la fin de la section 1.2.1 sur la faç on dont on pouvait imaginer calculer des points fixes.

Maintenant, $\bigvee_{n\in\mathbb{N}} f^n(\bot)$ existe toujours dans un treillis complet. Mais, alors que dans un treillis complet, les sups de n'importe quelle famille existent toujours, ici nous n'avons besoin que de sups de familles dirigées.

Définition 3 (Cpo) *Un* ordre partiel complet, *ou* cpo *("complete partial order") est un ensemble ordonné $(X, \leq)$ dans lequel toute famille dirigée a un sup. Un* cpo pointé *est un cpo ayant un élément minimal $\bot$.*

Corollaire 3 *Toute fonction continue f d'un cpo pointé X dans lui-même a un plus petit point fixe, qui est le sup des $f^n(\bot)$, $n \in \mathbb{N}$.*

On pourra s'interroger sur l'opportunité d'appeler une fonction f continue lorsqu'elle préserve les sups (vus comme limites). Il se trouve qu'il s'agit réellement de la notion de continuité usuelle vue en topologie générale.

Définition et lemme 4 (Topologie de Scott) *Soit* $(X, \leq)$ *un ensemble ordonné. Soit* $\mathcal{O}$ *l'ensemble des parties* O *de* X *telles que :*
- O *est* clos par le haut *: pour tout* x *dans* O*, si* $x \leq y$ *alors* y *est dans* O *;*
- *et* O *est* inaccessible par le bas *: pour toute famille dirigée* D *de* X *dont le sup existe et est dans* O*,* D *rencontre* O*, c'est-à-dire* $D \cap O \neq \emptyset$*.*

Alors $\mathcal{O}$ *est une topologie sur* X*, appelée la* topologie de Scott*. Les éléments de* $\mathcal{O}$ *sont appelés les* ouverts de Scott *de* X*.*

La propriété d'inaccessibilité par le bas dit, dans le cas où D est une suite croissante $(x_n)_{n\in\mathbb{N}}$, que si le sup (la "limite") des x_n est dans D, alors l'un des x_n est dans D. Par contraposée, toute suite croissante hors de O a un sup (limite) hors de O, ce qui est une faç on de dire que le complémentaire de O est fermé, dans un sens intuitif.

Démonstration. D'abord, la partie vide et X lui-même sont dans $\mathcal{O}$, clairement.

Ensuite, si $(O_i)_{i\in I}$ est une famille (possiblement infinie) d'ouverts de Scott, on prétend que $\bigcup_{i\in I} O_i$ est un ouvert de Scott : $\bigcup_{i\in I} O_i$ est clairement clos par le haut, et si D est une famille dirigée dont le sup existe et est dans $\bigcup_{i\in I} O_i$, alors $\bigvee D$ est dans un O_i, donc D rencontre O_i, ce qui entraîne que D rencontre $\bigcup_{i\in I} O_i$

Finalement, si O_1 et O_2 sont deux ouverts de Scott, montrons que leur intersection en est encore un. Clairement $O_1 \cap O_2$ est clos par le haut. Pour toute famille dirigée D dont le sup existe et telle que $\bigvee D$ est dans $O_1 \cap O_2$, alors $\bigvee D$ est dans O_1 et dans O_2, donc il existe deux éléments $x_1 \in D \cap O_1$ et $x_2 \in D \cap O_2$. Comme D est dirigée, x_1 et x_2 ont un majorant x dans D, qui est dans $O_1 \cap O_2$ parce que O_1 et O_2 sont clos par le haut. Donc D rencontre $O_1 \cap O_2$. □

▸ **EXERCISE 1.12**

Pour tout x dans X, notons $\downarrow x$ l'ensemble de tous les $x' \leq x$ dans X. Montrer que $\downarrow x$ est un fermé de Scott. (On rappelle qu'un *fermé* est par définition le complémentaire d'un ouvert, et non une partie non ouverte.)

▸ **EXERCISE 1.13**

Soit f une fonction monotone de l'ensemble ordonné $(X, \leq)$ dans l'ensemble ordonné $(Y, \leq)$. Démontrer que f est Scott-continue si et seulement si f est continue pour la topologie de Scott, autrement dit si et seulement si l'image réciproque de tout ouvert de Scott est encore un ouvert de Scott. (Pour la direction seulement si, on montrera d'abord que l'image d'une famille dirigée par une fonction monotone est dirigée. Pour la direction si, plus difficile, on considérera l'ensemble F, intersection des $\downarrow y$ lorsque y parcourt l'ensemble des majorants de $f(D)$, et on démontrera que F est un fermé tel que $f^{-1}(F)$ contient D.)

Dans la suite, on ne dira plus "Scott-continu", mais simplement "continu", l'exercice 1.13 montrant qu'il n'y a en fait aucune ambiguïté.

▸ **EXERCISE 1.14**
Montrez que la topologie de Scott sur $(X, \leq)$ n'est en général pas séparée (ou Hausdorff, ou T_2 : pour tous $x \neq x'$, il existe un ouvert O contenant x et un ouvert O' contenant x' d'intersection vide), en ce sens que si elle est séparée, alors tous les éléments de X sont incomparables pour $\leq$. Montrez en revanche que toute topologie de Scott est T_0, c'est-à-dire que pour tous $x \neq x'$, il existe un ouvert de Scott contenant x mais pas x', ou bien contenant x' mais pas x. (Utilisez l'exercice 1.12.)

▸ **EXERCISE 1.15**
Étant donné un espace topologique $(X, \mathcal{O})$, son *préordre de spécialisation* $\preceq$ est défini par : pour tous x, x' dans X, $x \preceq x'$ si et seulement si, pour tout ouvert O contenant x, O contient x'. Montrer qu'il s'agit bien d'un préordre, c'est-à-dire d'une relation réflexive et transitive. Si de plus la topologie $\mathcal{O}$ est T_0, alors il s'agit d'une relation d'ordre.

▸ **EXERCISE 1.16**
Montrer que l'ordre de spécialisation (cf. exercice 1.15) de la topologie de Scott d'un ensemble ordonné $(X, \leq)$ quelconque est l'ordre $\leq$ lui-même. (Pour l'un des deux sens de l'implication à démontrer, pensez à utiliser l'exercice 1.12.)

L'exercice suivant montre en quoi les sups sont une bonne faç on de parler de limites dans les cpos.

▸ **EXERCISE 1.17**
On dit que x est une *limite* de la famille dirigée D dans l'espace topologique X si et seulement si, pour tout ouvert O contenant x, il existe y dans D tel que pour tout $z \geq y$ dans D, z est dans O. Montrer que le sup de D s'il existe est une limite de D, pour la topologie de Scott. En vous aidant de l'exercice 1.16, montrer que le sup de D s'il existe est en fait la *plus grande limite* de D. (On rappelle à ceux que cette formulation étonnerait que le théorème d'unicité des limites n'est valable que dans les espaces séparés...)

2 Leç on 2

2.1 Langages impératifs, le langage C

Aujourd'hui nous examinons plus en détail les constructions de base d'un langage impératif typique : le langage C, inventé par Kernighan et Ritchie au début des années 1970.

La chose essentielle à retenir est qu'un programme C définit une *séquence* d'instructions élémentaires, qui sont des *affectations* : pour simplifier, les affectations sont des instructions de la forme ⟨variable⟩ `=` ⟨expression⟩`;`, qui ont pour effet de calculer la valeur de l'expression à droite du signe =, et de la ranger ensuite dans la variable à gauche du signe =. L'organisation des affectations en séquence se fait à l'aide d'une certain nombre de constructions du langage, que nous verrons en section 2.1.3. Cette présentation de C n'est pas complète, et on pourra consulter différentes ouvrages de référénce, par exemple *Le langage C*, par Kernighan et Ritchie, ou la référence `http://diwww.epfl.ch/w3lsp/teaching/coursC/` par exemple. Pour les plus courageux, on pourra aussi consulter la norme ISO/IEC 9899-1999 définissant le C dit "ANSI C" (le C officiel).

2.1.1 Affectations

Par exemple,

```
x = 3;
```

a pour effet de mettre l'entier 3 dans la variable nommée `x`. Un autre exemple est l'instruction

```
x = y+1;
```

qui récupère l'entier stocké dans la variable `y`, lui ajoute 1, et range le résultat dans la variable `x`.

Attention ! Le symbole = n'est *pas* le prédicat d'égalité que l'on rencontre usuellement en mathématiques. La signification typique de $x = 3$ en mathématique serait celle d'une valeur de vérité, valant vrai si x vaut 3, et faux sinon. Ici, `x = 3` ne teste pas du tout si `x` égale 3 ou non. En revanche, il est vrai qu'une fois l'instruction `x = 3` exécutée, le contenu de la variable `x` vaut effectivement 3. Un moyen de voir à quel point = n'est pas le prédicat d'égalité des mathématiques est de considérer une instruction telle que :

```
x = x+1;
```

qui récupère l'entier stocké dans la variable `x`, lui ajoute 1, et range le résultat de nouveau dans `x`. Donc, par exemple, si `x` contient l'entier 3 avant d'effectuer `x = x+1;`, `x` contiendra 4 après. En particulier, en aucun cas on n'aura `x` égal à `x`+1 après exécution de cette instruction. Mais bien sûr la valeur de `x` après cette instruction égale la valeur de `x` avant.

Cette ambiguïté, qui peut surprendre ceux qui sont habitués aux mathématiques, a été levée dans d'autres langages impératifs, comme Pascal, où l'on écrirait `x := x + 1`, en utilisant le symbole `:=` pour bien marquer qu'il s'agit d'une affectation et non d'un test d'égalité.

Si = est l'affectation, le test d'égalité en C est le symbol `==`. On écrira donc :

```
if (x==3) ...
```

pour tester si `x` vaut 3, et non

```
if (x=3) ...
```

qui, dans un langage un peu plus sûr que C, serait une erreur, et serait rejeté par le compilateur. Mais C l'accepte gaiement… le langage C est en fait très permissif : la sémantique de C précise en effet que toute affectation, par exemple `x=3`, peut aussi être vue comme une expression, et a pour valeur la valeur de son côté droit. Ici, la valeur de `x=3` est donc 3, ce qui permet d'écrire des expressions comme `y=x=3;`, qui range 3 dans `x` (`x=3`), puis retourne le résultat de l'affectation `x=3`, c'est-à-dire 3, et le range dans `y` : donc `y=x=3;` range 3 dans `x` et dans `y`.

La syntaxe des expressions de C est assez riche, et un aperç u (pas toutà fait complet, voir n'importe quel livre d'introduction à C pour l'ensemble complet) est donné en figure 3.

```
e ::= x | y | ma_variable | ...                        variables
    | 0 | 1 | 2 | ... | -1 | -2 | ... | 'a' | 'b' |
            ... | 3.1415926536 | ...                   constantes
    | e+e | e-e | e*e | e/e | e%e | -e                 opérations arithmétiques
    | e&e | e|e | e^e | e>>e | e<<e | ~e               opérations bit à bit
    | e==e | e<e | e<=e | e>e | e>=e                   comparaisons
    | e&&e | e||e | !e                                 et, ou, non logiques
    | *e | e[e] | &e                                   pointeurs, tableaux
    | e(e, ..., e)                                     appel de fonction
```

FIG. 3 – Un aperç u de la syntaxe des expressions de C

On ne décrira pas la sémantique des différentes expressions de C. Sur les entiers (qui sont les entiers machines, pas tous les entiers ! sur une architecture de machine 32 bits standard, les entiers sont ceux compris entre $-2^{31} = -2\ 147\ 483\ 648$ et $2^{31} - 1 = 2\ 147\ 483\ 647$), $e_1 + e_2$ calcule l'addition des valeurs de e_1 et e_2 (modulo 2^{32} sur une architecture 32 bits ; pour être précis, la valeur de $e_1 + e_2$ sur ces machines vaut l'unique entier compris entre -2^{31} et $2^{31} - 1$ qui est congru à la somme des valeurs de e_1 et e_2 modulo 2^{32}) ; $e_1 - e_2$ calcule la différence de e_1 et e_2 (modulo 2^{32} de nouveau sur une architecture 32 bits), $e_1 * e_2$ calcule le produit de e_1 et e_2 (modulo 2^{32}), e_1/e_2 calcule leur quotient et e_1`%`e_2 le reste de la division de e_1 par e_2 (modulo 2^{32}). Attention : e_1/e_2 ne calcule pas le résultat de la division de e_1 par e_2, mais sa partie entière (le quotient, donc). Par exemple, $2/3$ en C donne 0, pas $0.666666\ldots$; et 2`%`3 calcule le reste, ici 2.

En revanche, on peut aussi calculer en C sur des *nombres flottants* (abréviation de “nombres à virgule flottante”, qui dénote la faç on de les repésenter sur machine), qui sont des approximations finies de réels. Les opérations $+$, $-$, $*$ et $/$ sont alors les addition, soustraction, multiplication et division usuelles, à arrondi près. Par exemple, `2.0`   3.0/ donne vraiment `0.66666666666`. On remarquera que les nombres flottants sont écrits avec un point décimal ; en particulier, `2` et `2.0` sont deux objets différents : l'un est un entier machine, l'autre un flottant.

▸ **EXERCISE 2.1**
Que valent `1-2`, `15/7`, `15%7`, `1+(3*4)`, `(1+3)*4`, `1+3*4` en C ? Vous pouvez vous aider du compilateur `gcc`, en écrivant un programme de la forme :

```
#include <stdio.h>

int main ()
{
  printf ("La valeur de 1-2 est %d.\n", 1-2);
  return 0;
}
```

L'instruction `printf` ci-dessus a pour effet d'imprimer la valeur souhaitée : faire `man printf` pour avoir le modus operandi précis de `printf`.

Attention : ceci vous donnera la sémantique de `1-2` sur votre machine, pour votre compilateur particulier. N'en concluez pas nécessairement que la sémantique de C *en général* sera celle que vous aurez observée sur votre machine. Dans les exemples ci-dessus, cependant, les indications de votre machine seront fiables, et généralisables à toute machine (a priori).

▸ **EXERCISE 2.2**
Que valent `(-10)/7`, `(-10)%7` sur votre machine ? Ceci correspond-il à ce que vous attendiez ? Si q est le quotient et r le reste de la division de $a = -10$ par $b = 7$, a-t-on $a = bq + r$?

▸ **EXERCISE 2.3**
Que valent `10/(-7)`, `10%(-7)`, `(-10)/(-7)`, `(-10)%(-7)` sur votre machine ? Inférez-en la spécification du quotient et du reste sur votre machine : a/b et $a\%b$ calculent q et r tels que $a = bq + r$, $|r| < |b|\ldots$ et quelles autres propriétés déterminant q et r de faç on unique ?

▸ **EXERCISE 2.4**
Il y a aussi toujours des cas pathologiques. Que valent `1/0`, `(-1)/0`, `0/0` sur votre machine ? (Attention, ceci est *totalement dépendant* de votre compilateur et de votre machine : sur la mienne, le programme est interrompu par un message `Floating point exception`, alors même qu'il ne s'agit pas de calcul en flottant mais bien en entier.)

▸ **EXERCISE 2.5**
(Si votre machine a des entiers 32 bits en complément à deux, ce qui est le cas pour toutes les machines de l'ENS à la date d'écriture de ce document.) Rappelez-vous que le plus petit entier représentable en machine est $-2^{31} = -2\ 147\ 483\ 648$, et le plus grand est $2^{31} - 1 = 2\ 147\ 483\ 647$. L'opposé du plus petit entier représentable n'est donc pas représentable ! En expérimentant avec `gcc`, comment calcule-t-il l'opposé du plus petit entier représentable ? Que valent `(-2 147 483 648)/(-1)`,`(-2 147 483 648)%(-1)` ?

2.1.2 Tableaux, structures

J'ai un peu menti en section 2.1.1 en disant que les affectations étaient de la forme ⟨variable⟩ =⟨expression⟩ en C. Les côtés gauches peuvent en fait être plus généraux.

C’est notamment le cas avec les *tableaux*. En C, on peut déclarer un tableau, à l’entrée d’une fonction, par une déclaration de la forme

```
int a[50];
```

par exemple. Ceci déclare `a` comme étant un tableau de 50 éléments, chaque élément étant un entier machine (de type `int`). Alors que `int b` déclare `b` comme une variable contenant un seul entier, `a` ci-dessus en contient une rangée de cinquante.

On peut accéder à chaque élément du tableau en écrivant `a[`e`]`, où e est une expression retournant un entier. L’expression `a[0]` a pour valeur le premier élément du tableau, `a[1]` le deuxième, ..., `a[49]` le cinquantième (attention au décalage !). Les expressions `a[-1]`, `a[50]`, `a[314675]`, qui intuitivement dénoteraient des accès en-dehors du tableau, n’ont aucune sémantique : si l’on insiste pour obtenir leur valeur, on obtient en général n’importe quoi.

On peut aussi écrire des expressions d’accès aux éléments de tableaux plus compliqués. Par exemple, si `i` est une variable entière, `a[i]` dénote l’élément numéro `i` (qui est donc le `i`+1ième). L’intérêt de ceci est que l’on peut accéder à un élément variable, dont l’indice est lui-même calculé. Voici par exemple un calcul de produit scalaire de vecteurs à trois composantes :

```
float produit_scalaire (float a[3], float b[3])
{
  float resultat;
  int i;

  resultat = 0.0;
  for (i=0; i<3; i++)
    resultat = resultat + a[i]*b[i];
  return resultat;
}
```

(La boucle `for (i=0; i<3; i++)` itère l’instruction qui suit pour `i` allant de 0 inclus à 3 exclu.)

Pour changer le contenu des éléments d’un tableau, l’instruction d’affectation est en fait plus générale que celle que nous avons vue plus haut. On peut donc notamment écrire

```
a[0] = 5.0;
```

pour mettre le flottant `5.0` en premier élément du tableau (sans toucher aux autres). On peut ainsi effectuer une multiplication matricielle comme suit, par exemple :

```
  float a[3][3]; /* Multiplication de matrices 3x3, a et b,
                    résultat dans c. */
  float b[3][3]; /* A noter que les tableaux bidimensionnels
                    sont juste des tableaux à trois éléments
                    de tableaux à trois éléments. */
  float c[3][3];
```

```
  int i,j,k;

  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      {
        c[i][j] = 0.0;
        for (k=0; k<3; k++)
          c[i][j] = c[i][j] + a[i][k]*b[k][j];
      }
```

Les autres structures de données importantes en C sont les structures, les unions, et les pointeurs. Je ne parlerai pas des pointeurs ici, et je préfère attendre que vous ayez vu un peu d'architecture des machines au travers du langage machine en leç on 3 d'abord ; les pointeurs ne se comprennent vraiment qu'une fois qu'on sait comment la machine fonctionne vraiment, à un niveau plus bas.

Les *structures*, appelées aussi *enregistrements* ("records") sont essentiellement des n-uplets. Par exemple, on peut définir le type des nombres complexes en C comme

```
struct complex {
  float re;
  float im;
};
```

Ceci définit un nombre complexe comme étant un couple de deux flottants (sa partie réelle et sa partie imaginaire). Étant donnée une variable complexe `z`, typiquement déclarée par :

```
struct complex z;
```

on pourra accéder à sa partie réelle en écrivant `x.re`, et à sa partie imaginaire par `x.im`. Les identificateurs `re` et `im` sont appelés les *champs* de la structure `z`. De même que pour les tableaux, on pourra affecter (la valeur d')une expression à chaque champ individuellement. Par exemple,

```
struct complex x, y, z;

z.re = x.re + y.re;
z.im = x.im + y.im;
```

calcule dans `z` la somme des deux nombres complexes `x` et `y`.

▸ **EXERCISE 2.6**
Écrire de même la différence, le produit, et la division de deux nombres complexes.

▸ **EXERCISE 2.7**
Écrire un programme de multiplication de deux matrices 3×3 de nombres complexes (à compléter) :

```
struct complex a[3][3];
struct complex b[3][3];
struct complex c[3][3];
int i, j, k;

for (i=0; ...
```

Les structures C permettent de décrire des *produits cartésiens* de types. Si A_1, ..., A_n sont des types C, le type `struct toto {` A_1 `a1;` ...A_n `an; }` est essentiellement un type de n-uplets d'objets pris dans A_1, ..., A_n.

On peut aussi décrire quelque chose qui ressemble à des unions de deux types :

```
union nombre {
  int i;
  float x;
};
```

par exemple décrit le type `union nombre` comme consistant en des objets qui sont des entiers machine ou bien des flottants. Je vous déconseille d'utiliser cette construction avant de bien comprendre ce qu'elle fait vraiment : consultez n'importe quel livre d'introduction à C. Attention, cette construction n'a en fait pas grand-chose à voir avec l'union ensembliste, malgré les apparences. Elle permet d'autre part toute une série de hacks (et toute une série de bogues aussi) que je ne peux décemment pas vous recommander.

2.1.3 Structures de contrôle

Pour organiser les affectations en séquence, le langage C, comme la plupart des autres langages impératifs, dispose d'un certain nombre de constructions de base, étendues par des formes dérivées (du *sucre syntaxique*) :

- l'instruction qui ne fait rien (!) : elle se note `;`, ou bien `{}` ;
- la *composition séquentielle*, ou *séquence* : c_1`;`c_2`;` exécute l'instruction c_1, puis e_2. Par exemple, `x=3; y=x+1;` commence par ranger 3 dans `x` ; une fois ceci fait, ceci calcule `x+1`, soit 4, et le range dans `y` ;
- le *test*, aussi appelé la *conditionnelle* :

  `if (`e`)` c_1 `else` c_2

  commence par calculer la valeur de l'expression `e`, qui est censée être un entier machine. Cet entier représente une condition booléenne (vrai/faux). Si cet entier est non nul (ce qui par convention signifie que la condition est vraie), alors c_1 est executée, sinon c_2 est exécutée. Par exemple,

  ```
  if (x==3)
    printf ("x vaut bien 3.\n");
  else printf("Ah, x ne vaut pas 3, tiens.\n");
  ```

À noter ici que la sémantique de `x==3` n'est pas exactement de retourner vrai si `x` contient l'entier 3, et faux sinon ; en fait, `x==3` retourne l'entier 1 si `x` vaut 3, et 0 sinon. D'autres langages impératifs ont un type spécial (`bool` en Pascal, par exemple) pour dénoter les valeurs de vérité, C non : en C, les booléens sont codés sous forme d'entiers.

– La boucle *while* :

  $$\texttt{while (}e\texttt{)}\ c$$

  commence par calculer e. Si e est vrai (i.e., un entier non nul), alors c est exécutée. À la différence de la conditionnelle `if`, une fois l'exécutioin de c terminée, la boucle revient au début, recalcule e : si e est vrai de nouveau, c est reexécuté, et ainsi de suite, jusqu'à temps que e devienne faux (le cas échéant).

  On a déjà vu en leç on 1 que la boucleétait sémantiquement une conditionnelle plus un point fixe :

  $$\texttt{while (}e\texttt{)}\ c \;=\; \texttt{if (}e\texttt{)}\ \{c; (\texttt{while (}e\texttt{)}\ c)\ \}$$

Et c'est tout ! On peut calculer tout ce qui est calculable avec ces seules instructions.

C propose aussi quelques extensions syntaxiques. Notamment, le *bloc*

$$\begin{array}{l}\{\\ \quad c_1;\\ \quad \ldots\\ \quad c_n;\\ \}\end{array}$$

permet de voir n'importe quelle composition d'instructions $c_1, \ldots, c_n$ comme si ce n'était qu'une (grosse) instruction. Ceci permet notamment d'écrire :

```
if (x==3)
  {
    y = z;
    if (z==4)
      printf ("Non seulement x vaut 3 mais z vaut 4.\n");
    else printf ("z ne vaut pas 4, mais y vaut z maintenant.\n");
  }
else
  printf ("x ne vaut pas 3.\n");
```

On peut aussi éviter d'écrire la partie `else` d'un test dans certains cas : $\texttt{if (}e\texttt{)}\ c$ est une abréviation de $\texttt{if (}e\texttt{)}\ c\ \texttt{else ;}$.

Une autre construction très fréquente est la boucle *for* :

$$\texttt{for (}e_1; e_2; e_3\texttt{)}\ c$$

où e_1, e_2, e_3 sont des expressions (optionnelles : on peut ne rien écrire à la place de chacune), et c une instruction. Ceci est *exactement* équivalent à

$$e_1; \texttt{while (}e_2\texttt{)}\ \{\ c; e_3;\ \}$$

▸ **EXERCISE 2.8**
Réécrire la fonction `fact` du début de ce cours en terme de boucle *while*, comme indiqué ci-dessus. Ceci est-il lisible ?

▸ **EXERCISE 2.9**
L'idée de la construction `for` de C est de simuler une boucle pour `i` variant de m à n, comme la construction `for i=`a `upto` b `do`... du langage Pascal ou la construction `for i=`a `to` b `do ... done` de OCaml, en écrivant à la place `for (i=`a`; i<=`b`; i++) ...`. Cependant, en C, il est légal d'écrire :

```
for (i=1; i<=5; i++)
{
  printf ("La valeur de i est %d.\n", i);
  i = i+1;
}
```

Notez que la bizarrerie est que l'on change la valeur de `i` dans le corps de la boucle une seconde fois, en avant-dernière ligne. Que fait ce bout de code ? Peut-on faire pareil en OCaml ?

Finalement, en C on a aussi une construction `switch`. Disons en première approche que le code :

```
switch (e) {
      case ⟨constante 1⟩: c1; break;
      case ⟨constante 2⟩: c2; break;
      ...
      case ⟨constante n⟩: cn; break;
      default: c; break;
```

fait la même chose que (où `i` est une variable fraîche) :

```
{
 int i;
 i = e;
 if (i==⟨constante 1⟩)
    c1;
 else if (i==⟨constante 2⟩)
         c2;
 else if ...
     ...
 else if (i==⟨constante n⟩)
         cn;
 else c;
```

La construction `switch` est censée être plus rapide que la suite de `if` ci-dessus. Elle admet aussi quelques variantes, notamment, si l'on omet le mot-clé `break` en fin d'une ligne `case`⟨constante i⟩, l'exécution de c_i se continuera sur celle de c_{i+1}. C'est une source courante d'erreurs en C, et un des charmes particuliers du langage.

2.2 Langages fonctionnels, le cas de mini-Caml

Il existe aussi en C une notion de procédure que l'on peut appeler pour faire un calcul auxiliaire, et qui une fois le calcul fait retourne un résultat. C'est ce que nous avons déjà fait dans le programme `fact`.

Les langages *fonctionnels*, et notamment Caml et ses variantes (vous connaissez en général tous Objective Caml, aussi appelé OCaml) sont centrés sur cette notion de fonction. Il y a de nombreuses bonnes références sur OCaml, que l'on peut notamment trouver sur la page `http://cristal.inria.fr/`.

Nous allons plus particulièrement étudier une version réduite d'OCaml, que nous nommerons mini-Caml dans la suite. Il s'agit au passage du langage dont vous aurez à écrire un compilateur, puis un système d'inférence de types, en projet. Le typage vous sera enseigné par Delia Kesner. Dans la suite de ces notes, je considérerai une version non typée.

L'essentiel d'un langage fonctionnel est que l'on peut y écrire des fonctions : `fun` x `->` M, où M est une expression du langage, est encore une expression du langage, et dénote la fonction qui à x associe M. On appellera une telle construction syntaxique `fun` x `->` M une *abstraction*. Ceci est (pratiquement) la notion mathématique de fonction, comme on le verra quand on en donnera la sémantique un peu plus loin. Par exemple, `fun x -> x+1` est la fonction qui à tout entier n associe l'entier $n+1$.

On a d'autre part une construction, dite d'*application* : si M dénote une fonction et N dénote un argument possible de la fonction, alors MN dénote le résultat de l'application de la fonction M à l'argument N. En particulier, `(fun x -> x + 1) (3)`, que l'on peut aussi noter `(fun x -> x + 1) 3`, a bien pour valeur `4`. (On peut rajouter des parenthèses à volonté autour de sous-expressions pour éliminer les ambiguïtés syntaxiques, comme en C d'ailleurs.)

Le sous-langage de mini-Caml où l'on ne peut écrire que des variables, des abstractions et des applications s'appelle le λ-*calcul*, et a été inventé et étudié par Alonzo Church et ses successeurs à partir des années 1930. C'est aujourd'hui un outil fondamental en logique et en informatique théorique ; c'est au passage (pub !) le sujet du cours de logique et informatique du second semestre, en commun avec l'ENS de la rue d'Ulm.

En OCaml, on a en plus une construction permettant de donner des noms intermédiaires à des variables : `let` x`=`M `in` N a pour effet de calculer la valeur de M, de lui donner le nom x, puis de calculer N. Dans N, on a le droit de se référer à x pour dénoter la valeur de M. Au passage, ceci pourrait s'écrire autrement, sous forme de l'expression `(fun` x `->` N`)` M, ce serait équivalent… à part que le système de typage de Caml, que vous verrez avec Delia Kesner, considère la forme `let` d'une faç on un peu spéciale, et différente de la forme utilisant `fun` et l'application.

On retrouve, comme dans les expressions C, toute une famille de constantes et d'opérations permettant de faire des calculs : les constantes entières `0`, `1`, …, `-1`, `-2`, …, les sommes $M+N$, les différences $M-N$, les produits $M*N$, les quotients entiers (ou divisions flottantes) M/N, les restes M `mod` N, …, avec essentiellement la même sémantique qu'en C. (Nous considérerons que ces opérations ont réellement la même sémantique qu'en C. En OCaml, pour des raisons techniques, les entiers ne vont que de -2^{30} à $2^{30}-1$ sur une architecture 32 bits, et l'on calcule modulo 2^{31}, pas 2^{32}.)

Une différence importante avec C est que *la valeur des variables ne change pas*. En C, on peut toujours changer la valeur d'une variable, disons `x`, en écrivant par exemple `x=1`. Il n'y a aucune instruction permettant de changer la valeur d'une variable en OCaml ou en mini-Caml. Une conséquence est que, une fois calculée la valeur d'une variable, on peut être sûre qu'elle vaudra toujours la même valeur. Ceci est important pour la lisibilité des programmes. Dans un programme Caml de la forme `let x=3 in` N par exemple, on peut être sûr que toute référence à `x` dans N, aussi loin qu'on se trouve dans le source de la déclaration `let x=...`, vaudra `3`. En C, on peut écrire des instructions apparemment similaires :

```
{
  int x;

  x = 3;
  ... N ...
}
```

Mais pour être sûr que toutes les occurrences de `x` dans `N` se réfèrent bien à la valeur calculée `3`, il faudra d'abord s'assurer qu'aucune autre instruction n'a préalablement modifié la valeur de `x`. Ceci peut se révéler très compliqué.

Si OCaml et mini-Caml privilégient donc un style d'écriture mathématique, où les variables ne sont pas des cases où l'on range des valeurs comme en C, mais des noms dénotant des valeurs calculées par ailleurs, les langages de la famille Caml permettent cependant si on le souhaite d'écrire des affectations, plus ou moins comme en C. Pour ceci, on utilise la notion de *références*. En mini-Caml, l'expression `ref` M a pour effet d'*allouer* (de créer) une nouvelle référence, c'est-à-dire une nouvelle case mémoire. Cette case contient initialement la valeur de M. On peut ensuite relire le contenu d'une case mémoire (dénotée par N, disons) en écrivant l'expression `!`N. On peut aussi effectuer une affectation, et N`:=`P a pour effet de calculer la valeur de N, qui doit être une référence r, de calculer P, et de stocker la valeur de P à l'intérieur de la référence r.

Par exemple,

```
let x = ref 0
in !x
```

créer une référence contenant l'entier `0`, que l'on repère par le nom `x`, puis relit le contenu de cette référence : le résultat est `0`. Essayons une affectation :

```
let x = ref 0
in (x := 3; !x)
```

Ceci crée une référence contenant `0`, remplace son contenu par `3`, puis lit ce contenu : le résultat est `3`. Un peu plus compliqué :

```
let x = ref 3
in (x := !x+1; !x)
```

Ceci crée une référence contenant `3` ; l'effet de `x := !x+1` est d'ajouter un au contenu de cette référence, et le résultat final `!x` est donc `4`.

On a ici utilisé la construction `;` qui, comme en C, définit une composition séquentiel d'instructions (ou, comme ici, d'expressions). On dispose aussi d'une conditionnelle `if` M `then` N `else` P, qui calcule N si M est vrai et P sinon. (On considérera comme en C que seul l'entier 0 dénote le faux.) La valeur de la conditionnelle est celle de N dans le premier cas, celle de P dans le second. On notera que les conditionnelles ont une valeur en Caml, alors qu'elles étaient des instructions, qui n'ont pas de valeur, en C. (Pour dire la vérité, C dispose d'une autre instruction conditionnelle pour les expressions, dont la syntaxe est $e_1?e_2 : e_3$.)

On n'inclura pas de construction de boucle *while* en mini-Caml. L'exemple de la factorielle écrite en C et en Caml de la section 1.1 vous a peut-être suggéré que l'on pouvait coder la boucle *while* de la factorielle C en une construction `let rec` (définition de fonction récursive) de Caml. C'est effectivement le cas. Une construction `while (`e`)` c de C est grosso modo équivalente à la construction mini-Caml

```
letrec boucle = fun () ->
       if e
          then (c; boucle ())
       else ();;
boucle ()
```

qui définit d'abord un point fixe du `if` sous forme d'une fonction `boucle` à un argument (bidon), puis demande la valeur `boucle()` du point fixe.

La construction `letrec` f`=`M`;;` sera la seule construction de programme mini-Caml (les constructions vues plus haut sont des constructions d'expressions). Elle définit f comme valant exactement la même chose que M, et ce même si M fait appel à f. Pour des raisons techniques, on restreindra M à être elle-même la définition d'une fonction `fun` $x \to N$. Une définition via `letrec` (par exemple, pour `boucle`) de f est une *définition récursive*, c'est-à-dire que f est définie en fonction d'elle-même. Ceci signifie bien sûr que f doit être définie comme un point fixe de la fonction `fun` f `->` M.

La construction `letrec` de mini-Caml correspond au `let rec` d'OCaml. La construction `let` x`=`M`;;` en OCaml, qui définit x comme valant M *non récursivement* (M ne peut pas utiliser la valeur de x que l'on est en train de définir) correspond à `let` x`=`M`;;` en mini-Caml.

On termine cette description de l'essentiel d'OCaml et de mini-Caml en mentionnant que ces langages disposent d'une construction de produit cartésien : l'expression $(M_1, \ldots, M_n)$ dénote le n-uplet formé des valeurs respectives des expressions M_1, …, M_n dans cet ordre. D'autre part, l'expression $\texttt{proj}_1 M$ permettra d'extraire la première composante du n-uplet M (si c'en est un et si $n \geq 1$), $\texttt{proj}_2 M$ retrouvera la deuxième composante (si $n \geq 2$), et ainsi de suite. Il s'agit d'un vrai produit cartésien. En particulier, contrairement aux `struct` de C, il est impossible de changer après coup la valeur d'une des composantes d'un n-uplet.

On résume la syntaxe de mini-Caml en figure 4.

En guise d'exemple résumant notre discussion informelle de la sémantique de mini-Caml ci-dessus, voici une réalisation possible de `cat` en mini-Caml :

			Termes (fonctions, données)
$M, N, P, \ldots$	$::=$	x	variables (*non* modifiables)
	$\mid$	$M N$	application de M à N
	$\mid$	$\texttt{fun}\ x \to M$	fonction qui à x associe M
	$\mid$	$\texttt{let}\ x = M\ \texttt{in}\ N$	définition
	$\mid$	$\texttt{ref}\ M \mid !M \mid M{:=}N$	références (modifiables)
	$\mid$	$\texttt{0} \mid \texttt{1} \mid \ldots \mid M{+}N \mid \ldots$	arithmétique
	$\mid$	$(M_1, \ldots, M_n) \mid \texttt{proj}_i M$	n-uplets
	$\mid$	$M; N$	séquence
	$\mid$	$\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P$	tests
$Prog$	$::=$	ϵ	Programmes
	$\mid$	$Prog \quad \texttt{letrec}\ f = \texttt{fun}\ x \to M;;$	
	$\mid$	$Prog \quad \texttt{let}\ x = M;;$	

FIG. 4 – La syntaxe de mini-Caml

```
letrec boucle_interne = fun f ->
       let c = fgetc f
       in if c=EOF
             then 0
          else (putchar c; boucle_interne f);;
letrec boucle_externe = fun i -> fun argc -> fun argv ->
       if i<argc
          then let nom = sub argv i
            in let f = fopen nom "r"
            in (boucle_interne f; fclose f;
                boucle_externe (i+1) argc argv)
       else 0;;
letrec main = fun argc -> fun argv ->
       boucle_externe 1 argc argv;;
```

Ceci suppose que `fgetc` est, comme en C, une fonction prenant un fichier `f` en argument et retournant le prochain caractère lu dans ce fichier. De même, on suppose que `EOF` a été prédéfini comme la constante dénotant la fin de fichier (autrement dit, -1). On suppose aussi que `putchar` prend un caractère `c` et l'affiche, que `fopen` ouvre un fichier dont le nom est en premier argument et le mode d'ouverture (ici, `"r"`) est en second, que `fclose` ferme un fichier donné en argument. Autrement dit, on supposera qu'on a accès en mini-Caml à toutes les fonctions standard disponibles par ailleurs en C. (Ce sera le cas dans votre projet.) Finalement, on suppose que `sub argv i` récupère l'élément numéro `i` du tableau `argv`.

On pourra remarquer dans cet exemple le codage des fonctions à plusieurs arguments. On a choisi, comme il est traditionnel en Caml, de coder les fonctions à deux arguments comme des fonctions prenant leur premier argument, et retournant une nouvelle fonction qui prend le

deuxième argument et effectue le calcul. Par exemple, le point d'entrée main du programme est défini comme une fonction prenant un paramètre argc en argument, retournant une fonction prenant un paramètre argv en argument, puis appelant boucle_externe.

On aurait pu aussi définir les fonctions binaires comme des fonctions prenant des couples (M_1, M_2) en argument. Ce serait facile en OCaml, grâce à l'utilisation de *filtres d'entrée* ("patterns"), mais déjà moins commode en mini-Caml, qui se veut un langage relativement minimal.

3 Leç on 3

3.1 Architecture et assembleur

Nous allons maintenant regarder un peu plus en profondeur encore comment l'exécution d'un programme se passe sur une machine, en l'occurrence une machine à processeur Pentium et tournant sous Linux.

Reprenons l'exemple du programme `mycat` que nous avons déjà étudié en section 1.1. L'utilitaire `ddd` nous a permis de voir le programme s'exécuter, pas à pas (voir figure 2). Maintenant, lorsque `ddd` se lance, cliquez le menu "Source/Display Machine Code". Ceci vous montrera un cadre ressemblant à celui de la figure 5. Vous pouvez exécuter le code en pas à pas stop en cliquant sur le bouton "Next", ce qui vous fera avancer d'une instruction C à la fois, soit en cliquant sur le bouton "Nexti", ce qui vous fera avancer d'une instruction assembleur à la fois. Notez qu'une instruction C correspond en général à plusieurs instructions assembleur.

Vous pouvez aussi voir le contenu des principaux registres du Pentium en cliquant sur le menu "Data/Status Displays...", puis sur la case "List of integer registers and their contents" (pas sur "List of all registers..."). Les plus importants seront `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`, `%esp`, et `%eip`. Ce dernier est le *compteur de programme* (extended instruction pointer). Le registre `%esp` est le pointeur de pile, et `%ebp` est couramment utilisé pour sauvegarder la valeur de `%esp` en entrée de fonction. Les autres sont des registres à usage général. Tous contiennent des entiers 32 bits. Cette organisation des registres, ainsi que les instructions assembleur particulères que vous verrez sous `ddd` sont particulières au Pentium, mais le principe de l'assembleur est grosso modo le même sur toutes les machines.

3.1.1 Mémoires

Dans un ordinateur, on trouve d'abord une *mémoire*. Il s'agit d'un gigantesque tableau de cases contenant des *octets*, c'est-à-dire des nombres à 8 chiffres en base 2. (Un chiffre en base 2 est 0 ou 1, et est traditionnellement appelé un *bit*.) Les indices de ce tableau sont des nombres, typiquement de 0 à $2^{32} - 1 = 4\ 294\ 967\ 295$ sur le Pentium, qui code ces indices sur 32 bits. Les indices de ce tableau sont traditionnellement appelés les *adresses*.

On peut voir en figure 6 un exemple de mémoire d'un ordinateur. Comme le montre le tableau du haut de la figure 6, une mémoire n'est donc rien d'autre qu'une fonction qui à chaque adresse associe un contenu, qui est un octet. Il est traditionnel de compter les adresses et de lire les octets en *hexadécimal*, c'est-à-dire en base 16 ; la lettre `a` vaut 10, `b` vaut 11, ..., `f` vaut 15. Ainsi les adresses sont 0, 1, ..., 8, 9, a, b, c, d, e, f, 10, 11, ..., 1a, 1b, ..., 1f, 20, ... Lorsqu'il y aura ambiguïté dans la base, on utilisera la convention du langage C de faire précéder une suite de caractères parmi `0`, ..., `9`, `a`, ..., `f` des caractères `0x` pour montrer que le rester est un nombre hexadécimal. Par exemple, `0x10` dénote 16 ; ou bien `0xcafe` vaut $12 \times 256^3 + 10 \times 256^2 + 15 \times 256 + 14 = 51\ 966$.

La mémoire du haut de la figure 6 contient donc l'octet `0xca`$= 12 \times 16 + 10 = 202$ à l'adresse `0x0`$= 0$, l'octet `0xfe`$= 15 \times 16 + 14 = 254$ à l'adresse `0x1`$= 1$, et ainsi de suite.

Électriquement, une mémoire est un circuit ou un groupe de circuits ressemblant à celui en

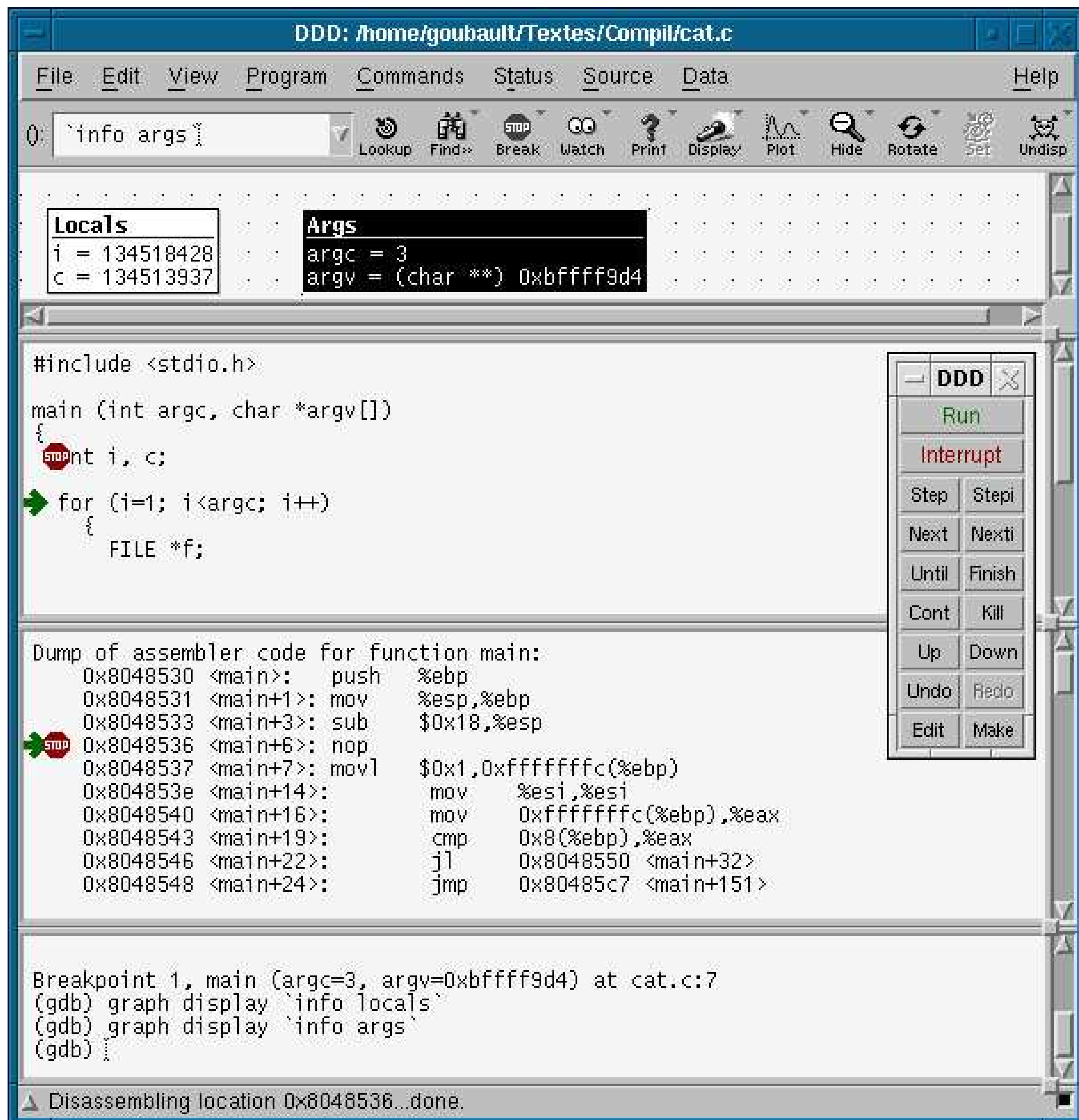
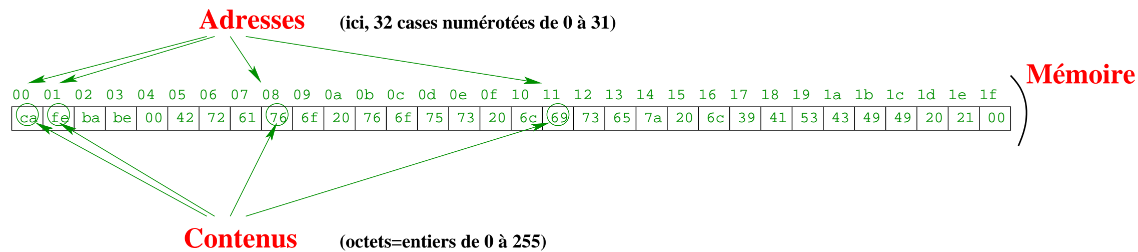


FIG. 5 – Une session sous ddd, montrant ensemble code assembleur et code C



Octets :

Hexa	Dec.	Binaire	ASCII
ca	202	11001010	Ê
fe	254	11111110	þ
76	118	01110110	v
69	105	01101001	i

Electronique :

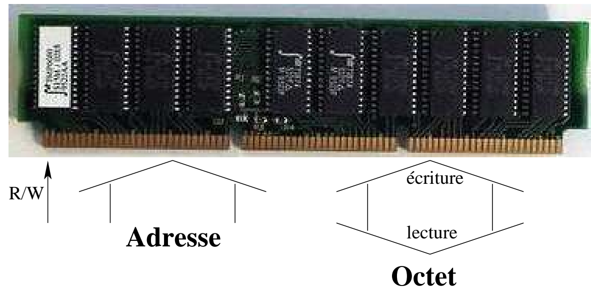


FIG. 6 – La mémoire des ordinateurs

bas à droite de la figure 6. Les adresses sont envoyées sous forme de signaux électriques (par exemple, si le bit i de l'adresse vaut 1 alors on relie le contact électrique no. i au +5V, sinon à la masse). Un autre contact, `R/W`, est positionné à 1 si l'on souhaite lire l'adresse ainsi décrite, auquel cas le contenu de l'octet à cette adresse sera présent en binaire sur huit autres contacts. Si `R/W` est positionné à 0, alors le circuit de mémoire s'attend en revanche à ce que la donnée à inscrire à l'adresse précisée soit déjà présente sur les huit contacts ci-dessus. Ceci est le principe général, il y a en fait d'autres contacts, et d'autres conventions (par exemple, le +5V peut en fait coder le 0 au lieu du 1, la tension peut ne pas être +5V, etc.)

Il est d'autre part important que tous les octets sont utilisés comme *codages*. L'octet `0xca` peut être vu comme :

- l'entier positif ou nul $12 \times 16 + 10 = 202$, comme ci-dessus (on dit que c'est un entier *non signé* ;
- ou bien l'entier compris entre $-2^7 = -128$ et $2^7 - 1 = 127$, et égal au précédent modulo $2^8 = 256$: ceci permet de représenter des entiers positifs ou négatifs dans un certain intervalle ; selon cette convention, `0xca` vaut alors $202 - 256 = -54$, et l'on parle d'*entier signé en complément à deux* ;
- ou bien le code du caractère "`Ê`" : il s'agit d'une pure convention, que l'on appelle la table ASCII (American Standard Code for Information Interchange), voir la figure 7 (source : `http://www.bbsinc.com/iso8859.html`). Par exemple, le caractère de code `0x43` se trouve dans la ligne `40`, colonne `3` : il s'agit du `C` ;
- ou bien l'instruction assembleur `lret` ("long return") du processeur Pentium ;
- ou bien encore pas mal d'autres possibilités...

Tout dépend de ce que l'on fait avec l'octet en question : des opérations arithmétiques sur entiers

signés, sur entiers non signés, des opérations de manipulations de chaînes de caractères, exécuter un programme, etc.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
80			‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ			
90		‘	’	“	”	•	–	—	˜	™	š	›	œ			Ÿ
A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

FIG. 7 – La table ASCII étendue

3.1.2 Le processeur

Le processeur (Pentium par exemple) est un petit automate, dont l'état interne est donné par le contenu de ses registres : on considérera que ceux du Pentium sont `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`, `%esp`, et `%eip`. Ce dernier est le compteur de programme.

À tout moment, le processeur est sur le point d'exécuter une instruction du *langage machine*. Il s'agit de l'instruction que l'on trouve en mémoire à l'adresse dont la valeur est le contenu du registre `%eip`. Dans le cas de la figure 8, le contenu du registre `%eip` et l'entier `0x08048530`, et l'on trouve en mémoire à cette adresse l'octet `0x55`, qui est le code de l'instruction `pushl %ebp` du Pentium.

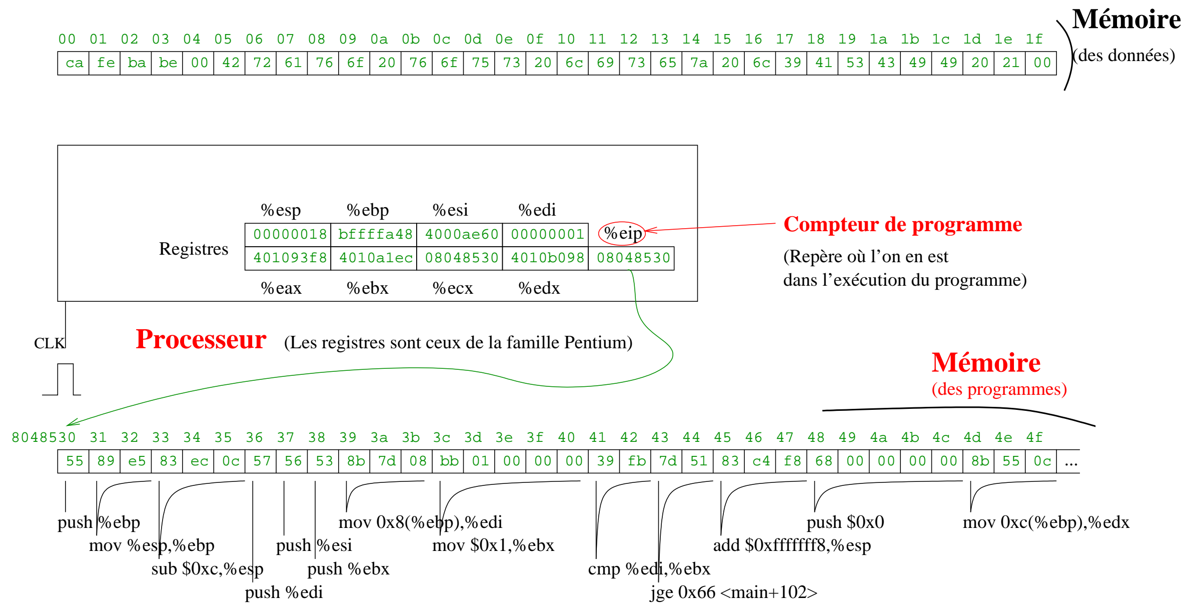


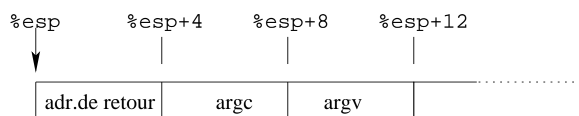
FIG. 8 – Le processeur et la mémoire

Le processeur est cadencé par un oscillateur, qui s'appelle l'*horloge*, et en quelque sorte bat la mesure : à chaque tic d'horloge (à peu de choses près), le processeur va chercher l'instruction suivante, l'exécute, puis attend le prochain tic d'horloge.

Dans l'exemple de la figure 8, au prochain tic d'horloge, le processeur va exécuter l'instruction `pushl %ebp`, qui a pour effet d'empiler ("push") la valeur du registre `%ebp` (ici `0xbffffa48`) au sommet de la pile. Quelle pile ? Eh bien, le Pentium, comme beaucoup d'autres processeurs, maintient dans le registre `%esp` ("extended stack pointer") l'adresse du sommet d'une pile. Dans l'exemple, le sommet de la pile est donc en ce moment à l'adresse `0x18`, et empiler `%ebp` a pour effet de stocker sa valeur `0xbffffa48` aux adresses `0x17`, `0x16`, `0x15`, `0x14`, et de bouger le pointeur de pile jusqu'en `0x14`. Le résultat est montré en figure 9. Les valeurs ayant changé sont en rouge. À noter qu'après l'exécution de l'instruction, le compteur de programme `%eip` est incrémenté et pointe vers l'instruction suivante.

L'instruction suivante est `movl %esp, %ebp`. La sémantique de cette instruction est de recopier (`movl`="move", déplacer) le contenu du registre `%esp` dans le registre `%ebp`. Au total, les deux premières instructions de ce programme ont sauvegardé le contenu précédent du registre `%ebp` sur la pile, puis ont mis l'adresse du sommet de la pile dans `%ebp`, pour pouvoir s'y référer dans la suite.

La raison de ce mouvement est que, à l'entrée de la fonction `main` (voir le source C en figure 1), la pile ressemble à ceci :



Le paramètre d'entrée `argc` est donc stocké 4 octets plus loin que le sommet de pile, sur 4 octets

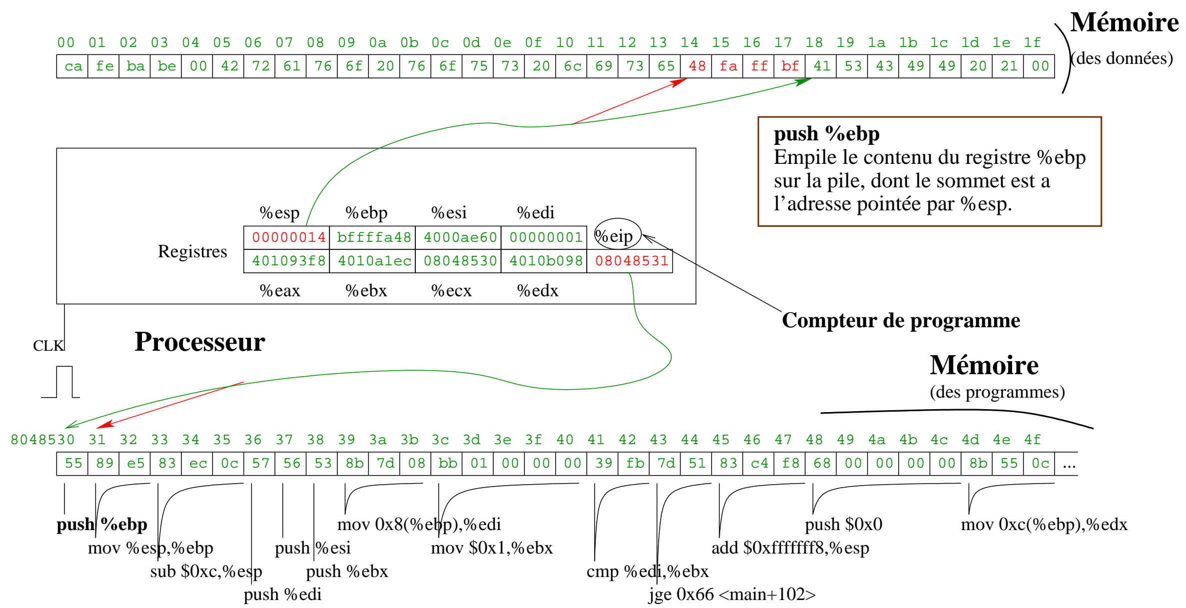


FIG. 9 – Exécution de la première instruction

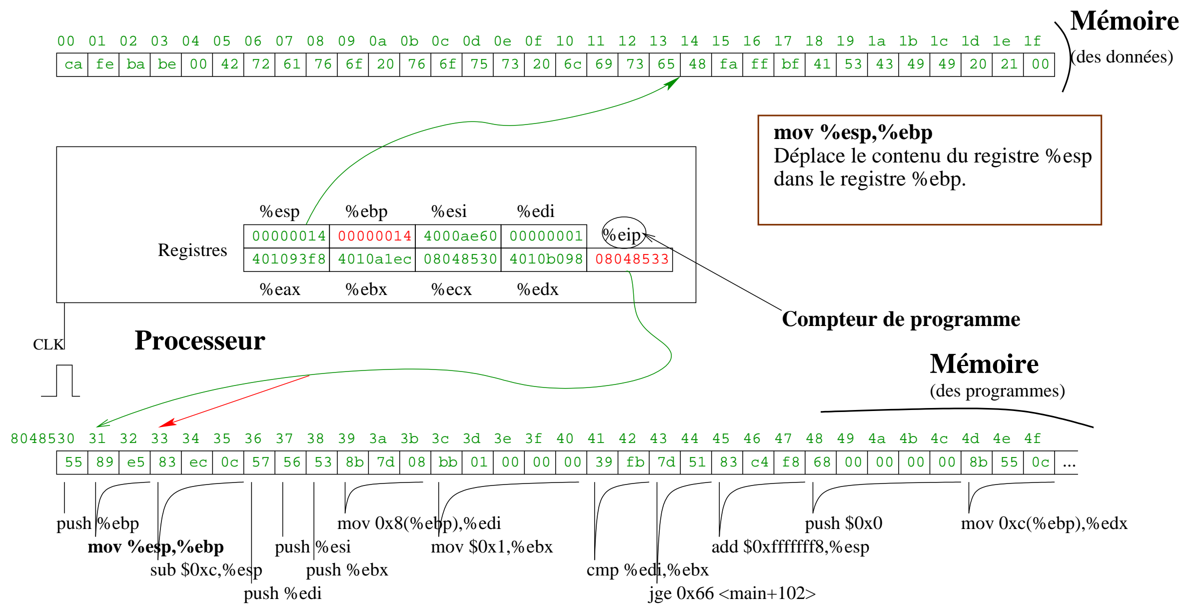
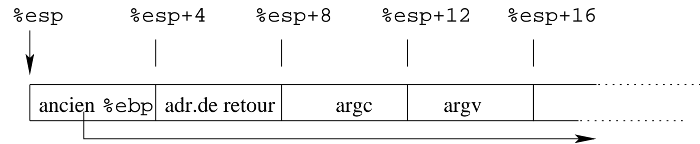


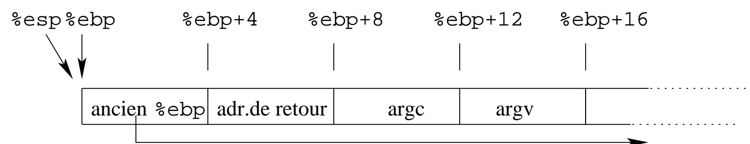
FIG. 10 – Exécution de la deuxième instruction

(32 bits), et le second paramètre d'entrée `argv` est stocké 8 octets au-delà du pointeur de pile, sur 4 octets aussi. Les 4 octets à partir du sommet de pile stockent l'*adresse de retour*, c'est-à-dire la valeur du compteur de programme où il faudra reprendre l'exécution lorsque l'exécution de la fonction `main` sera terminée.

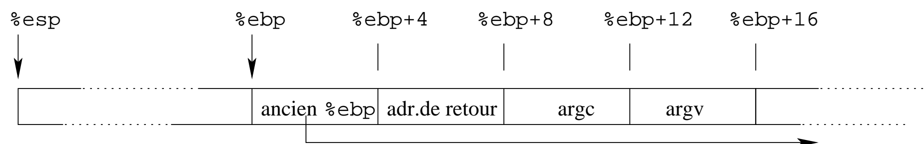
Après l'instruction `pushl %ebp`, la pile ressemble à :



Puis, après l'instruction `movl %esp, %ebp`, elle est de la forme :



L'instruction `subl $0xc, %esp` a pour but de retirer `0xc`= 12 du registre `%esp`. En d'autre termes, ceci réserve 12 octets sur la pile. On a donc une pile de la forme :

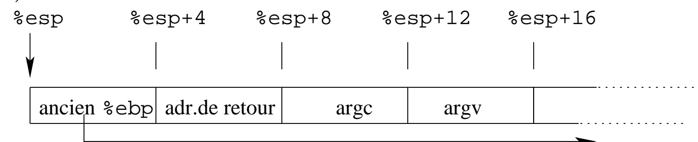


Concrètement, la situation est comme montrée sur la figure 11. Au lieu de soustraire 12 octets, on aurait pu en ajouter −12, ce qui se serait fait avec l'instruction `addl $-12, %esp`, ou de faç onéquivalente, `addl $0xfffffff4, %esp`. (Donc, que fait l'instruction `addl $0xfffffff8, %esp` un peu plus loin dans le programme ?)

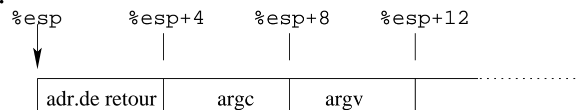
L'intérêt de l'allocation des 12 octets avant `%ebp` est de permettre de réserver de la place pour les trois variables `i`, `c` et `f` de la fonction `main`, typiquement.

Lorsqu'on arrivera à la fin de la fonction `main`, on y trouvera les trois instructions :

– `movl %ebp, %esp`, qui recopie le contenu du registre `%ebp` dans le registre `%esp`. Ceci a simplement pour effet de dépiler d'un coup tout ce qui avait été empilé au cours de la fonction `main`, et l'on revient à la situation :



– `popl %ebp`, l'instruction symétrique de `pushl %ebp`, qui dépile le sommet courant de la pile, et met la donnée 32 bits à l'ancien sommet de pile dans le registre `%ebp`. On revient à la situation :



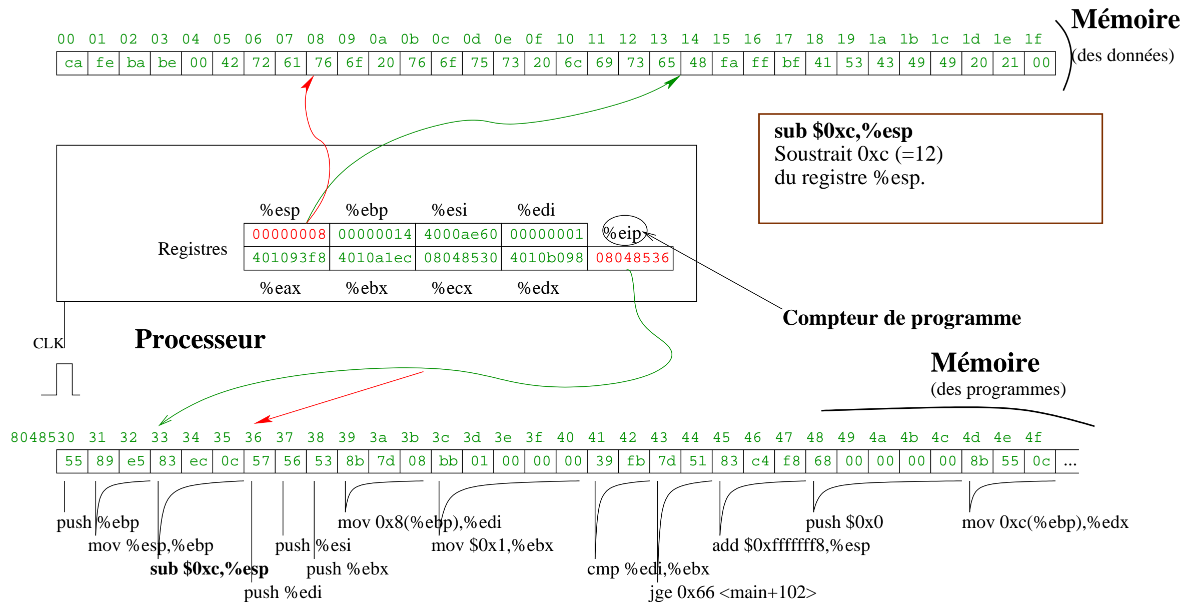


FIG. 11 – Exécution de la troisième instruction

– `ret`, qui retourne au programme appelant. Ceci dépile l'adresse au sommet de la pile, et s'y branche (autrement dit, met cette adresse dans le registre `%eip` de compteur de programme).

▸ **EXERCISE 3.1**
À l'aide de `ddd`, exécutez en pas à pas, avec l'instruction "Nexti", le programme `mycat` de la figure 1. Observez bien les contenus des registres. Vous pouvez aussi observer le contenu de la pile : regardez la valeur de `%esp` dans la liste des registres à l'entrée de la fonction `main`, disons `0xbffff8e0`, soustrayez disons 32 octets, ce qui donne `0xbffff9c0`, puis allez dans le menu "Data/Memory...", et demandez "View" "48" "hex" "words" "from : 0xbffff9c0" (remplacez par l'adresse réelle calculée plus haut). Déduisez-en ce que fait chaque instruction, si possible. Comparez avec la table de l'annexe A.

3.1.3 Modes d'adressage et formats d'instructions

Observons la différence entre les instructions `movl %esp, %ebp` et `subl $0xc, %esp`. Dans le premier cas, on recopie le contenu du registre `%esp` dans un autre registre : l'instruction `movl` recopie une donnée *source* vers une *destination*. Dans le second cas, on soustrait une donnée *source* du contenu de la *destination*.

Mais surtout, dans le premier cas, la source est un *registre*, à savoir le registre de sommet de pile `%esp`, alors que dans le second cas, il s'agit d'une constante (dite *immédiate*, parce que donnée, là, dans le code exécuté). Les modes typiques d'adressage sont :

– *immédiat* : la constante est donnée directement. La syntaxe correspondante dans l'assembleur Pentium consiste à faire précéder la constante par le signe `$`. Par exemple, `movl $0x14, %eax` met la constante `0x14`= 20 dans le registre `%eax`.

- *registre* : la donnée est dans un registre. Par exemple, `movl %eax, %ebx` recopie le contenu du registre `%eax` dans le registre `%ebx`.
- *absolu* : la donnée est à l'adresse donnée directement dans le code. Par exemple, `movl 0x8f90ea48, %e` recopie l'entier 32 bits stocké aux adresses `0x8f90ea48` à `0x8f90ea4b` (4 octets) dans le registre `%eax`.
- *indirect* : la donnée est à l'adresse que l'on trouve dans le registre spécifié. Par exemple, `movl (%ebx), %eax` récupère le contenu de `%ebx`, disons `0x8f90ea48`, puis lit l'entier 32 bits à cette adresse, c'est-à-dire de`0x8f90ea48` à `0x8f90ea4b` (4 octets), et le stocke dans le registre `%eax`.

Il y a quelques variantes de ces modes d'adressage. Dans le cas du mode absolu, il arrive que l'adresse ne soit pas stockée directement dans le code, mais sous forme d'un décalage entre la valeur courante de `%eip` (le compteur de programme) et l'adresse souhaitée : c'est l'adressage *relatif*, typiquement utilisé dans les instructions de saut (voir plus loin).

Le mode indirect connait aussi un certain nombre de déclinaisons de plus en plus raffinées. Par exemple, `movl 0x8(%ebx), %eax` ajoute d'abord l'*offset* (décalage) `0x8`$= 8$ à l'adresse stockée dans `%ebx` avant de récupérer la donnée qui s'y trouve. Dans l'exemple traité en section 3.1.2, `movl 0x8(%ebp), %eax` permet ainsi de récupérer la valeur de l'argument `argc`, une fois le prologue `pushl %ebp`, `movl %esp, %ebp` effectué ; de même, `movl %0xc(%ebp), %eax` permet de récupérer (l'adresse) du tableau `argv` dans `%eax`. Le Pentium dispose d'une profusion de tels modes. Notamment, `movl (%eax, %ebx, 4), %ecx` récupère dans `%ecx` le contenu de la mémoire aux 4 octets à partir de l'adresse obtenue en additionnant le contenu de `%eax` avec le contenu de `%ebx` multiplié par 4...

On a illustré ci-dessus l'utilisation des modes d'adressage dans le cas de la source. La destination peut être elle aussi spécifiée en mode registre, absolu, ou indirect. Elle ne peut pas être donnée en mode immédiat : on ne peut pas modifier la valeur d'une constante !

En général, toutes les combinaisons de modes d'adressage pour la source et la destination ne sont pas possibles. En théorie, l'idéal est de se référer à la documentation du processeur considéré, mais celle du Pentium est gigantesque. Il sera plus simple dans notre cas de compiler (avec `gcc`) quelques programmes assembleur simples : si `gcc` refuse, c'est que l'instruction n'existe pas.

Tous les processeurs disposent, en plus des instructions de déplacement (`movl` par exemple) et des instructions arithmétiques et logiques (`addl`, `subl`, le et bit à bit `andl`, etc.), des instructions *de saut*. Sur le Pentium, l'instruction `jmp` ⟨dest⟩ détourne le contenu du programme vers l'adresse ⟨dest⟩ :

- `jmp 0x8f90ea48` se branche à l'adresse `0x8f90ea48` (adressage absolu), autrement dit met l'entier `0x8f90ea48` dans le registre `%eip`.
- `jmp *%eax` se branche à l'adresse qui est stockée dans `%eax` (adressage indirect). Autrement dit, ceci recopie `%eax` dans `%eip`. (On notera une irrégularité de syntaxe ici : on n'écrit pas `jmp (%eax)`, même si cela aurait été plus cohérent.)

▸ **EXERCISE 3.2**

L'instruction `leal` ("load effective address") ressemble à `movl`. Mais là où `movl` ⟨source⟩, ⟨dest⟩ recopie ⟨source⟩ dans ⟨dest⟩, `leal` ⟨source⟩, ⟨dest⟩ recopie l'*adresse* où est stocké ⟨source⟩

dans ⟨dest⟩. Donc, par exemple, `leal 0x8f90ea48, %eax` (adressage absolu) est équivalent à `movl $0x8f90ea48, %eax` (adressage immédiat), et `leal 4(%ebx), %eax` met le contenu de `%ebx` plus 4 dans `%eax`, au lieu de lire ce qui est stocké à l'adresse qui vaut le contenu de `%ebx` plus 4. Que fait `leal (%eax, %esi, 4), %ebx` ? Pourquoi les modes d'adressage immédiat et registre n'ont-ils aucun sens pour la source de `leal` ? L'instruction `jmp` ⟨source⟩ est-elle équivalente à `movl` ⟨source⟩`, %eip` ou bien à `leal` ⟨source⟩`, %eip` ? (À ceci près que ces deux dernières instructions n'existent pas, car il est impossible d'utiliser le registre `%eip` comme argument d'aucune instruction. . .)

Pour effectuer des tests (les `if` de C ou de Caml), il suffit d'utiliser une combinaison de deux instructions, `cmpl` et une instruction de la famille des *sauts conditionnels*. Par exemple, l'instruction C

```
if (i==j)
  a;
else b;
```

correspondra au code assembleur suivant, en supposant que `i` est dans le registre `%eax` et `j` dans le registre `%ebx` :

```
    cmpl %ebx, %eax    ; comparer i (%eax) avec j (%ebx)
    jne _b             ; si pas égaux (not equal), aller exécuter b,
                       ; à l'adresse _b.
    ... code de a ...  ; sinon exécuter a
    jmp _suite         ; puis passer à la suite du code, après le test.
_b:
    ... code de b ...
_suite:
    ... suite du code ...
```

On constate que `jne` branche à l'adresse (ici _b) en argument si `i` n'est pas égal à `j`, et continue à l'instruction juste après le `jne` sinon.

A contrario, `je` brancherait à l'adresse donnée en argument si `i` était égal à `j` ; `jl` brancherait si `i` était strictement inférieur (less) que `j` en tant qu'entier signé, `jge` si plus grand ou égal (greater than or equal to), `jg` si plus grand strictement (greater), `jle` si plus petit ou égal (less than or equal to) ; `jb` (below), `jnb` (not below), `ja` (above), et `jna` (not above) sont les instructions correspondantes lorsque `i` et `j` sont des entiers *non* signés.

▸ **EXERCISE 3.3**

L'instruction `call` appelle un sous-programme (en C, un sous-programme, c'est une fonction). La seule différence avec l'instruction `jmp`, c'est que `call` empile d'abord le contenu de `%eip`, c'est-à-dire l'adresse qui est juste après l'instruction `call`. Donc `call` ⟨dest⟩, alors que le compteur de programme vaut, disons, N, est équivalent à `pushl $`n suivi de `jmp` ⟨dest⟩. Symétriquement, `ret` serait équivalente à `popl %eip` (si cette instruction existait) : que fait `ret`, concrètement ?

▸ **EXERCISE 3.4**
Montrer que `pushl` ⟨source⟩ est équivalente à `subl $4, %esp` suivi de `movl` ⟨source⟩`, (%esp)`. Proposer un équivalent de `popl` ⟨dest⟩.

▸ **EXERCISE 3.5**
Supposons que `%eax` contient l'entier 4 et `%ebx` contient l'entier -3 (en signé) $= 4\ 294\ 967\ 293$ (en non signé). Noter que le branchement `jle` sera pris après une instruction `cmpl %ebx, %eax`. Qu'en est-il dans les cas de `jg`, `jl`, `jge`, `ja`, `jna`, `jb`, `jnb` ?

Encore une fois, on rappelle que les instructions utiles du Pentium sont récapitulées en annexe A.

3.1.4 Formalisation

Il est possible, et même parfois utile, de donner une description formelle de la sémantique des instructions assembleur. La version que nous donnerons ici est outrageusement simplifiée (voir la documentation Intel : la sémantique réelle de `jmp` prend à elle seule deux pages !), mais essentiellement fidèle : vous n'avez pratiquement aucune chance en programmant de vous apercevoir que la sémantique réelle des instructions est plus compliquée que celle que je vais décrire ici.

D'abord, on doit définir la sémantique des modes d'adressage. Soit Ad l'ensemble de toutes les adresses valides sur le processeur considéré : c'est l'intervalle $[0, 2^{32} - 1]$ sur le Pentium. Considérons pour simplifier la description des modes d'adressage que les registres du processeur ont aussi des adresses, qui sont des constantes `%eax`, `%ebx`, etc., portant les noms des registres en question, qui sont deux à deux distinctes et hors de Ad. Soit Rg l'ensemble de ces constantes. On supposera que Rg contient, outre les noms des registres déjà mentionnés au début de la section 3.1, le nom `%eflags` ; ceci servira pour la sémantique de l'instruction `cmpl` et des sauts conditionnels.

Une *configuration* C est un couple $(mem, regs)$, où mem est une fonction totale de Ad vers l'ensemble $[0, 2^8 - 1]$ des octets, la *mémoire*, et $regs$ est une fonction totale de Rg vers l'ensemble $[0, 2^{32} - 1]$ des mots 32 bits. Notons $a.b.c.d$, où a, b, c, d sont des octets, le mot 32 bits $a + 256b + 256^2c + 256^3d$: c'est le nombre $abcd$ écrit en base 256, avec le chiffre de poids faible en premier. On notera aussi $mem(i..i + 3)$ le mot 32 bits $mem(i).mem(i + 1).mem(i + 2).mem(i + 3)$: si mem est une mémoire, $mem(i..i + 3)$ est juste le mot 32 bits que l'on peut lire à partir de l'adresse i.

Définissons la sémantique de ⟨source⟩ sous forme d'une fonction prenant une configuration C en entrée et retournant un entier 32 bits $[\![\langle\text{source}\rangle]\!]_{rd}\, C$, qui est celui que l'on trouve à l'endroit spécifié par C :

$$
\begin{array}{rcll}
[\![\$n]\!]_{rd}\,(mem, regs) & = & n & \text{adressage immédiat, si } n \in [0, 2^{32} - 1] \\
[\![n]\!]_{rd}\,(mem, regs) & = & mem(n..n + 3) & \text{adressage absolu, si } n \in Ad \\
[\![r]\!]_{rd}\,(mem, regs) & = & regs(r) & \text{adressage registre, si } r \in Rg \\
[\![n(r)]\!]_{rd}\,(mem, regs) & = & \multicolumn{2}{l}{mem(regs(r) + n..regs(r) + n + 3)} \\
 & & & \text{adressage indirect, si } n \in [-128, 127], r \in Rg
\end{array}
$$

Dans ce dernier cas, la plage $[-128, 127]$ pour l'offset n est arbitraire, et particulière au processeur. (Ici, c'est la plage utilisée par le Pentium.)

▸ **EXERCISE 3.6**
Supposons $Ad = [0, 2^{32} - 1]$. La notation $mem(n..n+3)$ n'a bien sûr aucun sens si $n \geq 2^{32} - 3$. D'après vous, quelle est la sémantique de l'adressage absolu à l'adresse n lorsque n vaut $2^{32} - 3$, $2^{32} - 2$, $2^{31} - 1$? Corriger de même la sémantique de l'adressage indirect.

▸ **EXERCISE 3.7**
Il y a en fait des instructions qui lisent non pas des entiers 32 bits, mais seulement 16 bits ou seulement 8 bits. Écrire les sémantiques correspondantes $[\![_]\!]^{16}_{rd}$ et $[\![_]\!]^{8}_{rd}$.

De même, on peut définir la sémantique de $\langle$dest$\rangle$ sous forme d'une fonction prenant une configuration C en entrée, un mot 32 bits N, et retournant la configuration $[\![\langle\text{dest}\rangle]\!]_{wr}(C)(N)$ résultant de l'écriture de N à l'adresse $\langle$dest$\rangle$, lorsque l'on part de la configuration C :

$$\begin{array}{rcll}
[\![n]\!]_{wr}(mem, regs)(N) & = & (mem[n..n+3 := N], regs) & \text{adressage absolu, si } n \in Ad \\
[\![r]\!]_{wr}(mem, regs)(N) & = & (mem, regs[r := N]) & \text{adressage registre, si } r \in Rg \\
[\![n(r)]\!]_{wr}(mem, regs) & = & (mem[regs(r) + n..regs(r) + n + 3 := N], regs) & \\
 & & & \text{adressage indirect, si } n \in [-128, 127], r \in Rg
\end{array}$$

On note ici $mem[n..n + 3 := N]$ l'unique mémoire mem' telle que $mem'(n..n + 3) = N$ et $mem'(i) = mem(i)$ pour tout $i \in Ad \setminus [n, n + 3]$. De même, $regs[r := N]$ est l'unique fonction de Rg vers $[0, 2^{32} - 1]$ qui à r associe N et à tout $r' \in Rg \setminus \{r\}$ associe $regs(r')$.

▸ **EXERCISE 3.8**
Corriger la sémantique $[\![_]\!]_{wr}$, dans l'esprit de l'exercice 3.6.

On aura aussi parfois besoin d'une troisième fonction $[\![_]\!]_{ea}$ qui retourne l'adresse où $[\![_]\!]_{rd}$ effectue une lecture et $[\![_]\!]_{wr}$ effectue une écriture, dans les cas où ces opérations de lecture et d'écriture s'effectuent en mémoire :

$$\begin{array}{rcll}
[\![n]\!]_{ea}(mem, regs) & = & n & \text{adressage absolu, si } n \in Ad \\
[\![n(r)]\!]_{ea}(mem, regs) & = & regs(r) + n & \\
 & & & \text{adressage indirect, si } n \in [-128, 127], r \in Rg
\end{array}$$

Ceci est utilisé dans l'instruction `leal` (voir annexe A), mais surtout dans les instructions de saut, `jmp` et autres.

On peut maintenant définir la sémantique des instructions assembleur. Le point important est qu'un programme assembleur décrit un *système de transitions*, c'est-à-dire un triplet (Q, Δ), où Q est un ensemble dit d'*états*, et $\Delta \subseteq Q \times Q$ est la *relation de transition*. (Ce n'est rien d'autre qu'un graphe orienté.) L'intuition sous-jacente à un système de transitions est qu'il s'agit d'une machine : à tout moment, la machine est dans un état $q \in Q$, et peut se déplacer (au prochain tic d'horloge) dans un état $q' \in Q$ tel que $(q, q') \in \Delta$, et continuer ainsi.

Ici, l'ensemble des états Q sera juste l'ensemble de toutes les configurations C. Le système de transitions est donc particulièrement gigantesque !

▸ **EXERCISE 3.9**
Combien y a-t-il de configurations dans le Pentium, en supposant que Rg est de cardinal 10 (8 registres de calcul, plus `%eip` et `%eflags`) ? Comparez au nombre d'électrons dans l'univers multiplié par la taille de l'univers en angströms, multiplié par la durée de vie de l'univers en picosecondes… multiplié essentiellement par n'importe quelle constante que vous trouverez dans un livre de physique. Quel est le nombre le plus grand ?

Il est bien sûr hors de question d'énumérer tous les états, et pour chacun, q, l'ensemble des q' tels que $(q, q') \in \Delta$. Pour décrire la relation de transition Δ du système, nous allons en profiter pour utiliser un formalisme que nous reverrons dans la suite, celui de *sémantique opérationnelle*.

Formellement, une sémantique opérationnelle est un ensemble de règles permettant de dériver des *jugements*. Dans celle qui nous intéresse ici, nous utiliserons des jugements de la forme $C \Longrightarrow C'$, où C et C' sont deux configurations. Les règles sont de la forme

$$\frac{J_1 \ldots J_n \quad Cond}{J}$$

et expriment que l'on peut déduire le jugement J à partir des jugements J_1, …, J_n, dès que la condition $Cond$ est vérifiée. J_1, …, J_n et $Cond$ sont les *prémisses* de la règle, et J est sa conclusion.

Une de nos premières règles, ici, sera celle qui donne la sémantique de l'instruction `movl` :

$$\frac{\begin{array}{l} C = (mem, regs) \quad \wedge \\ pc = regs(\texttt{\%eip}) \quad \wedge \\ mem(pc..pc + w - 1) = \text{“}\texttt{movl}\ \langle\text{source}\rangle, \langle\text{dest}\rangle\text{”} \quad \wedge \\ C' = [\![\langle\text{dest}\rangle]\!]_{wr}(C)([\![\langle\text{source}\rangle]\!]_{rd}(C))[\texttt{\%eip} := pc + w] \end{array}}{C \Longrightarrow C'}$$

Cette règle n'a aucun jugement en prémisse, juste une condition, conjonction de quatre conditions atomiques. Par convention, la notation "$\langle$instr$\rangle$" représente la suite d'octets correspondant à l'instruction assembleur $\langle$instr$\rangle$, et $mem(pc..pc+w-1)$ dénote la suite d'octets $mem(pc), mem(pc+1), \ldots, mem(pc + w - 1)$. Toutes les suites d'octets ne représentent pas nécessairement des instructions légales. En fait, les ensembles d'instructions des processeurs forment toujours un code préfixe ; ceci signifie que, étant donné une adresse pc donnée, il y a au plus une instruction légale représentée par une suite d'instructions commenç antà l'adresse pc. Ceci détermine en particulier de faç on unique la *largeur* w de l'instruction, qui est un entier valant au moins 1.

La règle ci-dessus s'applique donc si l'instruction stockée à l'adresse pc, qui est repérée par le contenu $regs(\texttt{\%eip})$ du registre `%eip`, est une recopie `movl` $\langle$source$\rangle$, $\langle$dest$\rangle$. Dans ce cas, $[\![\langle\text{source}\rangle]\!]_{rd}(C)$ récupère le mot 32 bits stocké à l'adresse $\langle$source$\rangle$, l'appel à $[\![\langle\text{dest}\rangle]\!]_{wr}(C)$ le stocke à l'adresse $\langle$dest$\rangle$. L'opération ...$[\texttt{\%eip} := pc+w]$ a pour effet de modifier la configuration ainsi obtenue en changeant le contenu `%eip` pour l'avancer w octets plus loin, c'est-à-dire sur l'instruction suivante. Rappelons que $f[x := v]$ est la fonction qui à x associe v, et à tout $y \neq x$ associe $f(y)$.

À noter qu'on aura aussi pu écrire la règle ci-dessus sous la forme plus compacte :

$$\frac{\begin{array}{l} pc = regs(\texttt{\%eip}) \quad \wedge \\ mem(pc..pc+w-1) = \text{“}\texttt{movl}\ \langle\text{source}\rangle, \langle\text{dest}\rangle\text{”} \end{array}}{(mem, regs) \Longrightarrow [\![\langle\text{dest}\rangle]\!]_{wr}\,(mem, regs)([\![\langle\text{source}\rangle]\!]_{rd}\,(mem, regs))[\texttt{\%eip} := pc + w]}$$

La sémantique de l'instruction `cmpl` mérite quelques explications préliminaires : `cmpl` ⟨source⟩, ⟨dest⟩ compare ⟨dest⟩ avec ⟨source⟩ (dans le sens inverse que l'on imaginerait), et calcule divers indicateurs booléens. Intuitivement, `cmpl` ⟨source⟩, ⟨dest⟩ calcule si ⟨source⟩=⟨dest⟩, si ⟨source⟩>⟨dest⟩, si ⟨source⟩<⟨dest⟩ (⟨source⟩ et ⟨dest⟩ étant vus comme entiers signés, ou bien comme entiers non signés), etc. Il collectionne ensuite tous ces résultats booléens dans le registre `%eflags`.

Formellement, assimilons les booléens vrai et faux aux entiers 1 et 0 respectivement. Soit $\mathbb{B} = \{0, 1\}$ l'ensemble des booléens. Si f est une fonction de $\{0, 1, ..., 31\}$ dans $\mathbb{B}$, l'*indicatrice* $\mathbf{1}_f$ est l'ensemble des i tels que $f(i) = 1$. Pour toute partie e de $\mathbb{N}$, soit $[e]$ l'entier $\sum_{n \in e} 2^n$: en particulier $[\mathbf{1}_f]$ est un mot 32 bits. Par exemple, si f est la fonction qui associe 1 exactement à 3 et à 4, alors $\mathbf{1}_f = \{3, 4\}$ et $[\mathbf{1}_f] = 2^3 + 2^4 = 24$: c'est le nombre qui, écrit en binaire, vaut 0000 0000 0000 0000 0000 0000 0001 1000, c'est-à-dire qui a exactement ses bits 3 et 4 positionnés à 1 (en comptant à partir de 0, de la droite vers la gauche).

Ce codage permet de stocker jusqu'à 32 résultats booléens de comparaisons. Essentiellement, `cmpl` ⟨source⟩, ⟨dest⟩ calcule tous les résultats de comparaisons possibles, correspondant à tous les branchements conditionnels possibles : égalité, inégalités strictes ou non, en signé ou non. Il stocke l'ensemble de booléens correspondant sous forme d'un entier dans `%eflags`. Les branchements conditionnels n'auront plus ensuite qu'à tester le bit correspondant dans le registre `%eflags`.

Bizarrement (pour des raisons électroniques et historiques plus que logiques), `cmpl` ⟨source⟩, ⟨dest⟩ calcule en fait d'autres conditions booléennes. Soit n un entier relatif quelconque (en particulier, pas nécessairement représentable sur 32 bits). On dit que n *déborde* 32 bits en *non signé* si et seulement si $n < 0$ ou $n \geq 2^{32}$. On dit que n *déborde* 32 bits en *signé* si $n < -2^{31}$ ou $n \geq 2^{31}$. Ces conditions reviennent à dire que n n'est pas représentable exactement sous forme d'un entier non signé, resp. signé, sur 32 bits. L'instruction de comparaison `cmp` ⟨source⟩, ⟨dest⟩ calcule ⟨dest⟩−⟨source⟩, tant en signé qu'en non signé, et calcule les conditions booléennes correspondant au test d'égalité du résultat avec 0, au test de négativité, de débordement, et de retenue (test si la différence est plus grande que $2^{32} - 1$).

Formellement, soient $Flg = \{\texttt{ZF}, \texttt{SF}, \texttt{OF}, \texttt{CF}\}$ un ensemble de constantes entières distinctes deux à deux entre 0 et 31. (Les noms de ces constantes signifient “zero flag”, “sign flag”, “overflow flag”, “carry flag”.) Le résultat de l'instruction `cmp` ⟨source⟩, ⟨dest⟩ consistera à mettre $[\mathbf{1}_f]$ dans le registre `%eflags`, pour une certaine fonction $f : Flg \to \mathbb{B}$ que nous décrivons ci-dessous. Notons que $[\![_]\!]_{rd}$ retourne un entier non signé. Soit $o(n)$ le booléen 1 si n déborde en *signé*, 0 sinon. Soit $s(n)$ le *bit de signe* de n, c'est-à-dire le booléen 1 si et seulement si $n \bmod 2^{32}$ est dans l'intervalle $[2^{31}, 2^{32}[$; autrement dit si le 32ème bit (le dernier représentable

sur 32 bits) de n vaut 1.

$$\frac{\begin{array}{l} pc = regs(\texttt{\%eip}) \quad \wedge \\ mem(pc..pc+w-1) = \text{“}\texttt{cmpl}\ \langle\text{source}\rangle,\ \langle\text{dest}\rangle\text{”} \quad \wedge \\ z = [\![\langle\text{dest}\rangle]\!]_{rd}\,(mem, regs) - [\![\langle\text{source}\rangle]\!]_{rd}\,(mem, regs) \quad \wedge \\ f = \{\texttt{ZF} \mapsto (z=0), \texttt{SF} \mapsto s(z), \texttt{OF} \mapsto o(z), \texttt{CF} \mapsto (z<0)\} \end{array}}{(mem, regs) \Longrightarrow (mem, regs[\texttt{\%eflags} := [\mathbf{1}_f], \texttt{\%eip} := pc+w])}$$

On dira que le *flag* `ZF` est mis si et seulement si $f(\texttt{ZF})$ est vrai dans la définition ci-dessus, et de même avec les autres flags `SF`, `OF`, `CF`. Donc le flag `ZF` est mis si et seulement si z vaut 0, c'est-à-dire si et seulement si ⟨source⟩=⟨dest⟩. Ceci permettra notamment de tester l'égalité de ⟨source⟩ avec ⟨dest⟩. Le flag `CF` permet, lui, de tester si ⟨dest⟩<⟨source⟩ (en non signé). L'utilisation des autres flags est un peu plus compliquée, et est laissée en exercice.

▸ **EXERCISE 3.10**
Pour tout entier n 32 bits non signé, c'est-à-dire entre 0 et $2^{32}-1$, soit $n^{\pm}$ l'entier n vu comme entier *signé* : par définition, $n^{\pm} = n$ si $0 \leq n < 2^{31}$, $n^{\pm} = n - 2^{32}$ si $2^{31} \leq n < 2^{32}$. Notez que $n^{\pm} = n$ modulo 2^{32}. Montrez que ⟨dest⟩<⟨source⟩ en *signé*, autrement dit $[\![\langle\text{dest}\rangle]\!]_{rd}\,(mem, regs)^{\pm} < [\![\langle\text{source}\rangle]\!]_{rd}\,(mem, regs)^{\pm}$ si et seulement si un exactement des flags `SF` et `OF` est mis dans la règle ci-dessus, autrement dit si et seulement si, posant $z = [\![\langle\text{dest}\rangle]\!]_{rd}\,(mem, regs) - [\![\langle\text{source}\rangle]\!]_{rd}\,(mem, regs)$ comme ci-dessus, $s(z) = 1$ et $o(z) = 0$, ou bien $s(z) = 0$ et $o(z) = 1$. (N'hésitez pas à faire une analyse détaillée de tous les cas possibles !)

La sémantique précise des instructions de saut conditionnel dépend donc uniquement du contenu du registre `%eflags`. Une nouveauté est qu'il y a *deux* règles pour décrire la sémantique des sauts conditionnels, une pour le cas où l'instruction prend le branchement, une pour le cas où elle ne le prend pas :

$$\frac{\begin{array}{l} pc = regs(\texttt{\%eip}) \quad \wedge \\ mem(pc..pc+w-1) = \text{“je}\ \langle\text{dest}\rangle\text{”} \quad \wedge \\ [\mathbf{1}_f] = regs(\texttt{\%eflags}) \quad \wedge \\ f(\texttt{ZF}) = 1 \quad \wedge \\ pc' = [\![\langle\text{dest}\rangle]\!]_{ea}\,(mem, regs) \end{array}}{(mem, regs) \Longrightarrow (mem, regs[\texttt{\%eip} := pc'])}$$

$$\frac{\begin{array}{l} pc = regs(\texttt{\%eip}) \quad \wedge \\ mem(pc..pc+w-1) = \text{“je}\ \langle\text{dest}\rangle\text{”} \quad \wedge \\ [\mathbf{1}_f] = regs(\texttt{\%eflags}) \quad \wedge \\ f(\texttt{ZF}) = 0 \end{array}}{(mem, regs) \Longrightarrow (mem, regs[\texttt{\%eip} := pc+w])}$$

▸ **EXERCISE 3.11**
Écrire les règles de sémantique opérationnelle pour les autres instructions de branchement `jne`, `ja`, `jna`, `jb`, `jnb`, `jge`, `jl`, `jg`, `jle`, ainsi que pour `jmp`. Dans le cas de `jge` et `jl`, on s'aidera de l'exercice 3.10.

3.2 La sémantique dénotationnelle (de mini-Caml)

Dans la tradition de ce cours d'alterner considérations pratiques et théoriques, revenons maintenant à un langage de plus haut niveau : mini-Caml, que nous avons déjà présenté en section 2.2.

Nous allons enfin définir la fonction de sémantique qui à tout programme mini-Caml associe une valeur dans un domaine Val à définir, et dont je parlais déjà en section 1.1.

Un des buts de cette construction sera de permettre de définir des traducteurs (*compilateurs*) d'un langage vers un autre, en l'occurrence mini-Caml vers l'assembleur, et de *démontrer* qu'ils sont corrects. Intuitivement, supposons que l'on a une fonction de sémantique dénotationnelle $[\![_]\!]_{\mathrm{caml}}$ qui à tout programme mini-Caml $\pi \in \mathbf{Caml}$ associe sa valeur $[\![\pi]\!]_{\mathrm{caml}} \in Val$, et une fonction de sémantique dénotationnelle $[\![_]\!]_{\mathrm{asm}}$ qui à tout programme assembleur $\alpha \in \mathbf{Asm}$ associe sa valeur $[\![\alpha]\!]_{\mathrm{asm}} \in Val$, un compilateur c est une fonction de $\mathbf{Caml}$ dans $\mathbf{Asm}$, et il est *correct* si et seulement si, intuitivement, le diagramme suivant commute :

$$\begin{array}{ccc} \mathbf{Caml} & \xrightarrow{c} & \mathbf{Asm} \\ & {\scriptstyle [\![_]\!]_{\mathrm{caml}}} \searrow & \downarrow {\scriptstyle [\![_]\!]_{\mathrm{asm}}} \\ & & Val \end{array} \tag{1}$$

(La réalité sera plus complexe... notamment parce que la sémantique de l'assembleur que nous avons décrite en section 3.1.4 n'est pas une sémantique dénotationnelle.)

Dans cette section, nous allons examiner une définition possible de la sémantique de mini-Caml. Le but de cette définition est de comprendre les programmes mini-Caml dans un esprit le plus proche des intuitions mathématiques. Notamment, nous voudrons voir les termes $\texttt{fun}\ x \to M$ comme de véritables fonctions mathématiques.

3.2.1 Domaines, le modèle $\mathbb{P}\omega$ de Plotkin

Rappelons que la syntaxe de mini-Caml est donnée en figure 4. Un domaine Val contenant toutes les valeurs possibles des programmes mini-Caml doit en particulier donner une sémantique aux fonctions $\texttt{fun}\ x \to M$, et donc contenir toutes, ou du moins suffisamment de fonctions de Val vers Val. On est tenté de demander que Val contienne $Val \to Val$, l'espace de toutes les fonctions de Val dans Val.

Or ici, on a un premier problème, énoncé au corollaire 6 du théorème de Cantor :

Proposition 5 (Cantor) *Soit X un ensemble quelconque. Il n'existe pas d'injection de $\mathbb{P}(X)$ dans X.*

Démonstration. Supposons $i : \mathbb{P}(X) \to X$ injective. On peut alors construire $r : X \to \mathbb{P}(X)$ telle que $r(i(A)) = A$ pour tout $A \in \mathbb{P}(X)$. Il suffit de prendre $r(a) = A$ si $a = i(A)$ (ce qui est bien défini car i est injective), et $r(a) = \emptyset$ par exemple si a n'est pas dans l'image de i. Posons A_0 l'ensemble des $a \in X$ tels que a n'est pas dans $r(a)$. Posons $a_0 = i(A_0)$. Si a_0 est dans A_0, alors par définition de A_0, a_0 n'est pas dans $r(a_0)$. Or $r(a_0) = r(i(A_0)) = A_0$, donc a_0 n'est pas dans A_0, contradiction. Donc a_0 n'est pas dans A_0. C'est-à-dire que a_0 n'est pas un $a \in X$ tel que a n'est pas dans $r(a)$. Donc a_0 est dans $r(a_0) = r(i(A_0)) = A_0$, contradiction encore. $\square$

Corollaire 6 *Soit Val un ensemble, et supposons qu'il existe une injection j de $Val \to Val$ dans Val. Alors Val est un singleton.*

Démonstration. D'abord Val est non vide, car j appliqué à la fonction identité de Val dans Val, est dans Val.

Supposons que Val contient au moins deux éléments, et soit donc T et F deux éléments distincts de Val. Pour tout $A \in \mathbb{P}(Val)$, on peut définir une fonction de Val dans Val par $\chi_A(a) = T$ si $a \in A$, $\chi_A(a) = F$ sinon. La fonction $i : \mathbb{P}(Val) \to Val$ qui à $A \in \mathbb{P}(Val)$ associe $j(\chi_A)$ est alors une injection, ce qui est impossible par la proposition 5. □

En particulier $Val \to Val$ ne peut pas être inclus dans Val, l'inclusion étant une injection. On se trouve alors devant une situation bête : le seul modèle qui ait un sens est celui qui n'a qu'une valeur, qui serait la valeur de tous les programmes. Il n'y aurait alors aucun moyen de les distinguer.

Heureusement, si au lieu de demander que Val soit juste un ensemble, on demande que ce soit un cpo (voir section 1.2.3), et que les fonctions de Val dans Val soient continues (au sens de Scott), on va réussir à éviter ce dilemme. L'utilisation des cpo dans le cadre de la sémantique dénotationnelle a de plus une justification philosophique cohérente : une valeur v dans un cpo est un "calcul partiel", c'est-à-dire une donnée qui donne une indication sur ce qu'on cherche à calculer, mais à laquelle il peut manquer encore des portions. Plus l'on grimpe dans le cpo, plus la valeur devient précise, jusqu'à atteindre la valeur finale du programme, qui est le sup de toutes les valeurs partielles. C'est d'ailleurs par cette intuition que Dana Scott en est venu à inventer les cpo dans les années 1960–1970.

▸ **EXERCISE 3.12**
Il semble naturel de considérer, selon cette intuition, qu'une valeur *totale*, c'est-à-dire complètement calculée, soit par définition un élément maximal d'un cpo. Montrer, en utilisant l'axiome du choix que, pour tout cpo $(Val, \leq)$, pour tout $x \in Val$, il existe au moins une valeur totale au-dessus de x.

Notons $[Val \to Val']$ l'espace de toutes les fonctions continues de Val dans Val', muni de l'ordre point à point : $f \leq g$ si et seulement si $f(x) \leq g(x)$ pour tout $x \in Val$. Nous allons remplacer l'exigence que Val contienne $Val \to Val$ (l'espace de toutes les fonctions de Val dans Val) par le fait qu'il doive contenir seulement $[Val \to Val]$. Ceci correspond à une autre intuition de Scott, qui est que toutes les fonctions calculables sont continues.

D'autre part, nous pouvons demander que Val ne contienne $[Val \to Val]$ qu'à isomorphisme près ; dans les cpo, un isomorphisme est une fonction continue, bijective, et d'inverse continu. Notons $\cong$ la relation d'isomorphisme entre cpo. Il est facile de voir que le fait que Val contienne $[Val \to Val]$ à isomorphisme près est équivalent à demander l'existence de deux fonctions continues $i : [Val \to Val] \to Val$ et $r : Val \to [Val \to Val]$ telles que $r \circ i$ soit l'identité sur $[Val \to Val]$. (On dit alors que $[Val \to Val]$ est un *rétract* de Val.)

L'une des réussites de Scott est d'avoir démontré :

Théorème 7 (Scott) *Soit Val_0 un cpo quelconque. Alors il existe un cpo Val tel que $Val \cong [Val \to Val]$ et contenant Val_0 (à isomorphisme près).*

La construction est un peu compliquée, et je lui préférerai une construction moins générale, mais plus simple, le modèle $\mathbb{P}\omega$ de Gordon Plotkin. Notre but, rappelons-le, est de fournir un cpo non-trivial contenant son espace de fonctions continues.

Considérons l'ensemble $\mathbb{P}\omega$ des parties de $\mathbb{N}$. (La notation ω est une autre écriture pour $\mathbb{N}$, mais le nom traditionnel de ce modèle est $\mathbb{P}\omega$, pas $\mathbb{P}(\mathbb{N})$.) Muni de l'ordre d'inclusion $\subseteq$, il forme un ordre partiel… et même un treillis complet. C'est donc, en particulier, un cpo. Ce cpo est exactement le cpo Val que nous cherchons.

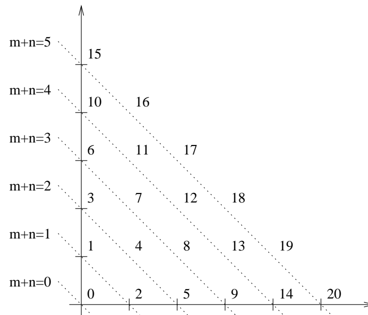


FIG. 12 – Codage des paires d'entiers

La construction du modèle $\mathbb{P}\omega$ est fondée sur quelques remarques. D'abord, on peut représenter toute paire[1] (m, n) d'entiers par un entier $\langle m, n\rangle$, par exemple par la formule :

$$\langle m, n\rangle = \frac{(m+n)(m+n+1)}{2} + m$$

La valeur de $\langle m, n\rangle$ en fonction de m en abscisse, et de n en ordonnée, est donnée en figure 12.

Ensuite, on peut représenter tout ensemble fini $e = \{n_1, \ldots, n_k\}$ d'entiers par le nombre binaire $[e] = \sum_{i=1}^{k} 2^{n_i}$. (Oui, nous avons déjà vu ce codage en section 3.1.4 !) Réciproquement, tout entier m peut être écrit en binaire, et représente l'ensemble fini e_m de tous les entiers n tels que le bit numéro n de m est à 1 : voir la figure 13, où l'on a écrit l'ensemble $\{1, 3, 4, 7, 9, 10\}$ sous la forme du nombre binaire 11010011010, soit 1690 en décimal — autrement dit, $[\{1, 3, 4, 7, 9, 10\}] = 1690$ et $e_{1690} = \{1, 3, 4, 7, 9, 10\}$.

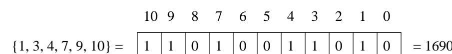


FIG. 13 – Codage des ensembles finis

▸ **EXERCISE 3.13**

Trouver une formule pour $\texttt{proj}_1 : \langle m, n\rangle \mapsto m$ et pour $\texttt{proj}_2 : \langle m, n\rangle \mapsto n$.

L'astuce principale dans la construction de $\mathbb{P}\omega$ est que la continuité des fonctions de $\mathbb{P}\omega$ dans $\mathbb{P}\omega$ permet de les représenter comme limites d'objets finis :

[1] On dit couple en franç ais, mais "pair" en anglais. Pour vous habituer au vocabulaire informatique, j'ai choisi de conserver l'anglicisme, qui est plus usuel dans le vocabulaire courant.

Lemme 8 *Pour toute fonction f de $\mathbb{P}\omega$ dans $\mathbb{P}\omega$, f est continue si et seulement si pour tout $e \in \mathbb{P}\omega$, $f(e)$ est l'union de tous les $f(y)$, y partie finie de e.*

Démonstration. Seulement si : supposons f continue, et soit e un ensemble non vide. L'ensemble des parties finies de e est dirigé, et son sup (l'union de ses éléments) est e. Donc $f(e)$ est le sup (l'union) de tous les $f(y)$, y partie finie de e.

Si : supposons que $f(e) = \bigcup_{y \text{ finie } \subseteq e} f(y)$, pour toute partie e de $\mathbb{N}$. Soit $(e_i)_{i \in I}$ une famille dirigée de parties de $\mathbb{N}$. Le sup des $f(e_i)$ est l'union des $f(y)$, lorsque y parcourt les parties finies d'au moins un e_i, alors que $f(\bigcup_{i \in I} e_i)$ est l'union des $f(y)$ lorsque y parcourt les parties finies de $\bigcup_{i \in I} e_i$. Clairement toute partie finie d'un e_i est aussi une partie finie de $\bigcup_{i \in I} e_i$. Réciproquement, si y est une partie finie de $\bigcup_{i \in I} e_i$, alors y est une partie finie d'une union finie de tels e_i : pour chaque $n \in y$, posons e_{i_n} un e_i tel que $n \in e_i$, alors y est inclus dans $\bigcup_{n \in y} e_{i_n}$. Comme $(e_i)_{i \in I}$ est dirigée, tout union finie de e_i est inclus dans un e_j, $j \in I$, donc si y est une partie finie de $\bigcup_{i \in I} e_i$, alors y est une partie finie d'au moins un e_j. Donc f est continue. □

L'idée est alors que toute fonction continue est définie par ses valeurs sur les ensembles finis, et que les ensembles finis sont codables par des entiers. On définit donc :

$$i : [\mathbb{P}\omega \to \mathbb{P}\omega] \to \mathbb{P}\omega \qquad f \mapsto \{\langle m, n \rangle \mid n \in f(e_m)\}$$

et son inverse à gauche :

$$r : \mathbb{P}\omega \to [\mathbb{P}\omega \to \mathbb{P}\omega] \qquad e \mapsto (x \mapsto \{n \in \mathbb{N} \mid \exists y \text{ fini } \subseteq x \cdot \langle [y], n \rangle \in e\})$$

Théorème 9 *Les fonctions r et i sont bien définies, continues, et $r \circ i$ est l'identité sur $[\mathbb{P}\omega \to \mathbb{P}\omega]$.*

Démonstration. Il est clair que i est bien définie ; r l'est à condition que $r(e)$ soit bien continue pour tout e. Or, si x est le sup d'une famille dirigée $(x_i)_{i \in I}$, autrement dit $x = \bigcup_{i \in I} x_i$, alors $r(e)(x) = \{n \in \mathbb{N} | \exists y \text{ fini } \subseteq \bigcup_{i \in I} x_i \cdot \langle [y], n \rangle \in e\}$; ceci vaut exactement $\{n \in \mathbb{N} | \exists i \in I \cdot \exists y \text{ fini } \subseteq x_i \cdot \langle [y], n \rangle \in e\} = \bigcup_{i \in I} r(e)(x_i)$, par un argument similaire à la deuxième partie de la démonstration du lemme 8.

Que r soit continue signifie que, si e est le sup d'une famille dirigée $(e_i)_{i \in I}$, alors le sup de la famille de fonctions $(x \mapsto \{n \in \mathbb{N} \mid \exists y \text{ fini } \subseteq x \cdot \langle [y], n \rangle \in e_i\})_{i \in I}$, c'est-à-dire la fonction qui à x associe $\{n \in \mathbb{N} \mid \exists i \in I \cdot \exists y \text{ fini } \subseteq x \cdot \langle [y], n \rangle \in e_i\}$ (on rappelle que l'ordre sur $[\mathbb{P}\omega \to \mathbb{P}\omega]$ est l'ordre point à point) doit être exactement la fonction qui à x associe $\{n \in \mathbb{N} \mid \exists y \text{ fini } \subseteq x \cdot \langle [y], n \rangle \in e\}$. Mais ceci est évident, parce que les quantificateurs existentiels commutent.

La continuité de i revient à dire que, si $(f_j)_{j \in I}$ est une famille dirigée de fonctions continues, alors $i(\bigvee_{j \in I} f_j) = \bigvee_{j \in I} i(f_j)$, c'est-à-dire $\{\langle m, n \rangle | n \in \bigcup_{j \in I} f_j(e_m)\} = \bigcup_{j \in I} \{\langle m, n \rangle | n \in f_j(e_m)\}$, ce qui est évident puisque les deux côtés de l'égalité valent $\{\langle m, n \rangle | \exists j \in I \cdot n \in f_j(e_m)\}$.

Finalement, vérifions que $r \circ i$ est l'identité sur $[\mathbb{P}\omega \to \mathbb{P}\omega]$. Soit f une fonction continue de $\mathbb{P}\omega$ vers $\mathbb{P}\omega$:

$$\begin{array}{rcl} r(i(f))(x) & = & \{n \in \mathbb{N} \mid \exists y \text{ fini } \subseteq x \cdot \langle [y], n \rangle \in i(f)\} \\ & = & \{n \in \mathbb{N} \mid \exists y \text{ fini } \subseteq x \cdot n \in f(y)\} \\ & = & \displaystyle\bigcup_{y \text{ fini } \subseteq x} f(y) \end{array}$$

Mais ceci vaut exactement $f(x)$, par le lemme 8. □

▸ EXERCISE 3.14

Un élément *fini* (aussi dit *compact*) d'un cpo Val est un élément x tel que, pour toute partie dirigée D telle que $\bigvee D \geq x$, alors il existe un élément y de D tel que $y \geq x$. Montrer que les éléments finis de $\mathbb{P}\omega$ sont précisément les parties finies de $\mathbb{N}$ (au sens usuel du mot "fini").

▸ EXERCISE 3.15

Notons, dans tout cpo Val, $\uparrow x = \{y \in Val | y \geq x\}$. Montrer que $\uparrow x$ est ouvert dans la topologie de Scott si et seulement si x est un élément fini de Val. La topologie engendré par les $\uparrow x$, x fini dans Val, est donc une topologie moins fine que la topologie de Scott. Montrer, en considérons le cpo $([0,1], \leq)$, que cette topologie est en général strictement moins fine ; on commencera par se demander quels sont les éléments finis de $([0,1], \leq)$.

▸ EXERCISE 3.16

Un cpo est dit *algébrique* si et seulement si tout élément est sup d'une famille dirigée d'éléments finis. Montrer que $\mathbb{P}\omega$ est un cpo algébrique. Montrer que, dans tout cpo algébrique, la topologie de Scott est *exactement* celle qui est engendrée par les $\uparrow x$, x fini. Le cpo $([0,1], \leq)$ est-il algébrique ?

4 Leç on 4

Nous continuons aujourd'hui nos efforts dans le but de décrire une sémantique aussi naturelle que possible de mini-Caml. Nous commenç ons par poursuivre la leç on 3, en donnant une sémantique dénotationnelle de mini-Caml en section 4.1. Nous verrons une sémantique déjà plus concrète en section 5.1, avec laquelle nous ferons le lien. Cette dernière sera une étape dans le processus qui nous permettra de définir un compilateur pour mini-Caml.

4.1 Sémantique dénotationnelle de mini-Caml

Nous allons définir une fonction $[\![_]\!]_{\text{caml}}$ qui prend une expression ou un programme mini-Caml en entrée, et retourne sa valeur. Supposons que Val est un cpo contenant $[Val \to Val]$, à isomorphisme près, par exemple $\mathbb{P}\omega$.

Une expression mini-Caml dépend de variables, et l'on ne peut pas décider de la valeur des variables comme ç a, ex abrupto. La fonction $[\![_]\!]_{\text{caml}}$ va donc prendre un argument supplémentaire, un *environnement* $\rho \in Env = Var \to Val$, où Var est l'ensemble de toutes les variables. L'environnement dit, pour chaque variable x, quelle est la valeur $\rho(x)$ que nous lui donnons. La valeur d'une expression, par exemple `x+1`, sera interprétée dans un tel environnement : si $\rho(\mathtt{x}) = 3$, alors intuitivement la valeur $[\![\mathtt{x} + 1]\!]_{\text{caml}} \rho$ de `x+1` dans cet environnement sera 4.

Une expression mini-Caml dépend aussi de l'état courant de la mémoire… Oh, pas encore la mémoire au sens où nous l'avons vue en section 3.1.1, mais quelque chose de plus abstrait et en même temps de proche dans l'esprit. Le but sera de donner une sémantique aux expressions portant sur les références, `ref` M, $!M$, $M{:=}N$.

Donnons-nous un ensemble *Addr* fini ou dénombrable quelconque. On appellera conventionnellement les éléments de *Addr* les *adresses*, et ceci devrait vous rappeler la section 3.1.1. Mais ici, nous allons être moins concrets : nous ne stockerons pas des octets à chaque adresse, mais des valeurs quelconques de notre cpo Val. De plus, nous allons considérer qu'il peut exister des adresses inutilisées, au sens où rien n'y est stocké. Une *mémoire* sera donc ici une fonction partielle de *Addr* vers Val. Mieux, on va même demander que les mémoires μ soient les éléments de l'ensemble $Mem = Addr \to_{fin} Val$ des fonctions *de domaine fini* de *Addr* vers Val. Ceci représentera qu'on ne peut stocker qu'une quantité finie d'information à tout moment.

Au total, la sémantique d'une expression mini-Caml M *dans un environnement* $\rho \in Env$ *et une mémoire* $\mu \in Mem$ sera un élément $[\![M]\!]_{\text{caml}} \rho\mu$ de $Mem \times Val$: une paire (μ', V) formée de la mémoire *après* l'exécution de l'expression, et de la valeur V retournée.

▸ **EXERCISE 4.1**
Montrer que, si Val est un cpo, alors Env et Mem aussi, pour l'ordre point à point.

4.1.1 Quel genre de cpo nous faut-il ?

Au vu des constructions de mini-Caml, on n'aura pas besoin d'un domaine Val qui contienne $[Val \to Val]$ (pour les fonctions), mais quelque chose ressemblant à $[Mem \times Val \to Mem \times Val]$: les fonctions mini-Caml démarrent elles-mêmes dans une certaine mémoire, et retournent une nouvelle mémoire. Ce ne sera pas suffisant. D'abord, il se peut qu'une telle fonction boucle,

et ne retourne jamais. Pour représenter ce comportement, on va considérer qu'une fonction mini-Caml n'a pas pour valeur une fonction continue de $Mem \times Val$ vers $Mem \times Val$, mais vers $(Mem \times Val)_\perp$, où $A_\perp$ dénote le cpo A avec un nouvel élément $\perp$ ajouté en-dessous de tous les éléments de A :

Définition 4 (Relèvement) *Pour tout cpo A, soit $A_\perp$ le cpo union disjointe de A et de $\{\perp\}$, avec $x \leq y$ si et seulement si $x = \perp$ ou $x, y \in A$ et $x \leq y$ dans A.*

Vu sous l'angle mathématique, $[Mem \times Val \to Mem \times Val]$ est bien un cpo, mais il n'est en général pas pointé. Or le corollaire 3 demande à ce que l'on se place dans un cpo pointé pour garantir l'existence de points fixes, et nous aurons besoin de tels points fixes pour définir la sémantique de la construction `letrec` $f =$ `fun` $x \to M;;$. Le cpo $[Mem \times Val \to (Mem \times Val)_\perp]$, lui, est pointé, et nous l'utiliserons comme domaine sémantique des fonctions mini-Caml.

▸ **EXERCISE 4.2**
Montrer que $[Mem \times Val \to (Mem \times Val)_\perp]$ a un plus petit élément. Lequel est-ce ?

▸ **EXERCISE 4.3**
Soit A un sous-cpo de $\mathbb{P}\omega$, c'est-à-dire un sous-ensemble de $\mathbb{P}\omega$ qui, muni de l'ordre ($\subseteq$) de $\mathbb{P}\omega$, est un cpo. Montrer que l'ensemble formé, d'une part, des parties de la forme $\{0\} \cup \{n+1 | n \in x\}$ avec $x \in A$, d'autre part de l'ensemble vide, est un sous-cpo de $\mathbb{P}\omega$ isomorphe à $A_\perp$.

▸ **EXERCISE 4.4**
Soient A et B deux sous-cpo de $\mathbb{P}\omega$. Montrer que $i([A \to B])$, l'image par i du cpo $[A \to B]$, est un sous-cpo de $\mathbb{P}\omega$ isomorphe à $[A \to B]$; i et r ont été introduites en section 3.2.1. Indication : surtout n'utilisez pas la définition de i et de r, juste leurs propriétés de continuité et le fait que $r \circ i$ est l'identité.

Ensuite, mini-Caml permet de raisonner sur des références. Intuitivement, la valeur d'une référence sera une adresse, dans $Addr$. Nous aimerions donc pouvoir considérer que Val contient $Addr$, mais Val est un cpo et $Addr$ un ensemble sans structure d'ordre particulière. On utilise alors l'astuce suivante :

Lemme 10 *Pour tout ensemble X, X muni de l'égalité comme ordre partiel est un cpo. On dit qu'un tel cpo est* plat.

Autrement dit, $Addr$ vu comme cpo plat est un cpo dont tous les éléments sont deux à deux incomparables. Ceci correspond bien à l'intuition de Scott que les éléments d'un cpo sont des valeurs partielles : dans un cpo plat, toute valeur est totale.

▸ **EXERCISE 4.5**
Rappelons que $Addr$ est fini ou dénombrable. Montrer que $Addr$ est isomorphe à un sous-cpo de $\mathbb{P}\omega$. Montrer que $\mathbb{B} = \{0, 1\}$, $\mathbb{N}$, $\mathbb{Z}$ sont isomorphes à des sous-cpo de $\mathbb{P}\omega$.

Mini-Caml permet aussi de raisonner sur des n-uplets. On a donc besoin d'une notion de produit cartésien de cpo :

Lemme 11 *Pour tous cpo A_1, ..., A_n, le* produit $A_1 \times \ldots \times A_n$ *est l'ensemble des n-uplets $(a_1, \ldots, a_n)$, $a_1 \in A_1$, ..., $a_n \in A_n$, avec l'ordre composante par composante :*

$$(a_1, \ldots, a_n) \leq (a'_1, \ldots, a'_n) \text{ ssi } a_1 \leq a'_1 \text{ et } \ldots \text{ et } a_n \leq a'_n$$

Il s'agit d'un cpo. De plus, les projections $\pi_i : A_1 \times \ldots \times A_n \to A_i$ sont continues. Enfin, si $f_1 : C \to A_1$, ..., $f_n : C \to A_n$ sont des fonctions continues, alors la fonction qui à $c \in C$ associe $(f_1(c), \ldots, f_n(c))$ est continue.

▸ **EXERCISE 4.6**
Démontrer le lemme 11.

▸ **EXERCISE 4.7**
Soient A_1, ..., A_n des sous-cpo de $\mathbb{P}\omega$. Posons $\langle\rangle = 0$, $\langle a \rangle = a$, et pour $k \geq 2$, $\langle a_1, \ldots, a_k \rangle = \langle a_1, \langle a_2, \ldots, a_k \rangle\rangle$. Montrer que $\{\langle a_1, \ldots, a_n \rangle | a_1 \in A_1, \ldots, a_n \in A_n\}$ est un sous-cpo de $\mathbb{P}\omega$ isomorphe à $A_1 \times \ldots \times A_n$.

De même que l'on dispose des produits, on a aussi les sommes, c'est-à-dire l'équivalent des unions *disjointes* :

Lemme 12 *Soit $(A_i)_{i \in I}$ une famille de cpo. Leur* somme*, ou* coproduit $\sum_{i \in I} A_i$ *est l'ensemble des couples (i, a) où $i \in I$ et $a \in A_i$, avec l'ordre somme défini par :*

$$(i, a) \leq (i', a') \text{ ssi } i = i' \text{ et } a \leq a' \text{ dans } A_i$$

Il s'agit d'un cpo. De plus, les injections $\iota_i : A_i \to \sum_{j \in I} A_j$ sont continues. Enfin, si $f_i : A_i \to C$ sont une famille de fonctions continues de A_i dans un cpo C, alors la fonction de $\sum_{i \in I} A_i$ dans C qui à (i, a) associe $f_i(a)$ est continue.

▸ **EXERCISE 4.8**
Démontrer le lemme 12.

▸ **EXERCISE 4.9**
Soit I un ensemble d'entiers. Soit $(A_i)_{i \in I}$ une famille de sous-cpo de $\mathbb{P}\omega$. Montrer que $\{\langle i, a \rangle | i \in I, a \in A_i\}$ est un sous-cpo de $\mathbb{P}\omega$ isomorphe à $\sum_{i \in I} A_i$.

▸ **EXERCISE 4.10**
Généraliser le lemme 11 et l'exercice 4.7 aux cas de produits infinis de cpo indexés par I. En quoi ceci diffère-t-il (ou non) d'un espace de fonctions continues de I vers un certain cpo ?

Avec le produit et la somme, on peut définir le cpo Val^* des *listes* (ou *mots*, ou *suites*) d'éléments de Val :

$$Val = \sum_{i \in \mathbb{N}} \underbrace{Val \times \ldots \times Val}_{i \text{ fois}}$$

On peut aussi définir les sommes finies $A_1 + \ldots + A_n$, qui ne sont autres que $\sum_{i \in I} A_i$, avec $I = \{1, \ldots, n\}$.

Nous avons besoin d'un domaine Val qui contienne :

- $Mem \times Val \to (Mem \times Val)_\perp$ pour représenter les fonctions ;
- $Addr$ pour les références ;
- Val^* pour les n-uplets ;
- $\mathbb{Z}$ pour les entiers ;
- $\mathbb{B}$ pour les booléens.

De plus, on souhaite pouvoir faire la différence entre fonctions, références, n-uplets, etc. Nous serons donc intéressé par la somme de tous ces cpo.

Dans les sections qui suivent, nous supposerons donc que Val est un cpo tel que, à isomorphisme près :

$$\begin{aligned} Val \supseteq\ & [Mem \times Val \to (Mem \times Val)_\perp] \\ & + Addr \\ & + Val^* \\ & + \mathbb{Z} \\ & + \mathbb{B} \end{aligned} \tag{2}$$

▸ **EXERCISE 4.11**

En utilisant les exercices 4.3, 4.4, 4.5, 4.7, 4.9, montrer que $Val = \mathbb{P}\omega$ est une solution à l'inégalité ci-dessus.

4.1.2 Sémantique dénotationnelle des expressions

Le problème du domaine des valeurs Val étant réglé, passons à la sémantique des expressions. Nous allons définir la quantité $[\![M]\!]_{\text{caml}}\, \rho\mu \in (Mem \times Val)_\perp$ par *récurrence structurelle* sur l'expression M, c'est-à-dire que $[\![M]\!]_{\text{caml}}\, \rho\mu$ sera définie en fonction de données $[\![N]\!]_{\text{caml}}\, \rho\mu$, où N est une sous-expression directe de M.

Les esprits attentifs auront au passage remarqué que le résultat de $[\![M]\!]_{\text{caml}}\, \rho\mu$ n'est finalement pas dans $Mem \times Val$, mais dans $(Mem \times Val)_\perp$: il est en effet possible que, en présence de `letrec`, l'évaluation de M ne termine pas.

Ceci étant dit, la sémantique des variables est immédiate :

$$[\![x]\!]_{\text{caml}}\, \rho\mu \quad = \quad (\mu, \rho(x)) \tag{3}$$

Une variable a en effet pour valeur ce que l'environnement dit, et la mémoire est inchangée.

Pour la sémantique des fonctions et de leurs applications, on utilise le fait que (à isomorphisme près), $[Mem \times Val \to (Mem \times Val)_\perp]$ est un sous-cpo de Val :

$$\begin{aligned} [\![MN]\!]_{\text{caml}}\, \rho\mu \ &=_{def}\ \phi([\![N]\!]_{\text{caml}}\, \rho\mu') \quad \text{si } \phi \in [Mem \times Val \to (Mem \times Val)_\perp], \quad (4) \\ &\qquad\qquad \text{où } (\mu', \phi) =_{def} [\![M]\!]_{\text{caml}}\, \rho\mu \neq \perp \\ &\qquad\qquad \text{et } [\![N]\!]_{\text{caml}}\, \rho\mu' \neq \perp \\ &=_{def}\ \perp \qquad \text{sinon} \\ [\![\texttt{fun}\ x \to M]\!]_{\text{caml}}\, \rho\mu \ &=_{def}\ (\mu, \phi) \qquad\qquad (5) \\ &\qquad\qquad \text{où } \phi \text{ est défine par } \phi(\mu'', V) = [\![M]\!]_{\text{caml}}\, (\rho[x \mapsto V])\mu'') \end{aligned}$$

En d'autres termes, pour calculer la valeur de MN dans l'environnement ρ, et en partant de la mémoire M, on calcule la valeur $[\![M]\!]_{\text{caml}}\,\rho\mu$ de M. Ceci peut valoir $\bot$ (pas moyen d'obtenir la valeur de M en tant que fonction), auquel cas la valeur de MN est $\bot$ de nouveau. Ou bien ceci vaut un couple (μ', ϕ) formé d'une nouvelle mémoire après l'évaluation de M, et d'une valeur ϕ. Si ϕ n'est pas une fonction, c'est-à-dire pas un élément de $[Mem \times Val \rightarrow (Mem \times Val)_\bot]$, encore une fois la valeur de ϕ est $\bot$ (l'indéfini). Sinon, on applique la fonction ϕ à la valeur de N, si pas indéfinie. La sémantique de $\texttt{fun}\ x \rightarrow M$ devrait être plus déchiffrable. Notez cependant que la mémoire μ ne change pas lors de l'exécution de $\texttt{fun}\ x \rightarrow M$: le couple retourné est (μ, ϕ), avec le même μ qu'en paramètre. Mais la valeur ϕ elle-même est une fonction qui prendra l'état courant de la mémoire au moment où on l'appliquera, et pourra retourner une nouvelle mémoire.

▸ **EXERCISE 4.12**
Intuitivement, dans quel ordre la définition de $[\![MN]\!]_{\text{caml}}\,\rho\mu$ force-t-elle l'évaluation de MN ?

▸ **EXERCISE 4.13**
Montrer que les côtés droits des équations 4 et 5 sont bien définis. Notez en particulier que les injections canoniques sont toutes continues. Vérifiez soigneusement en particulier que la fonction ϕ de l'équation 5 est bien continue, en supposant que $[\![M]\!]_{\text{caml}}\,\rho'\mu'$ est une fonction continue en $\rho' \in Env$ pour tout $\mu' \in Mem$.

La construction suivante de mini-Caml est le `let` :

$$\begin{aligned} [\![\texttt{let}\ x = M\ \texttt{in}\ N]\!]_{\text{caml}}\,\rho\mu &= [\![N]\!]_{\text{caml}}\,(\rho[x \mapsto V])\mu' \\ &\quad \text{si } (\mu', V) = [\![M]\!]_{\text{caml}}\,\rho\mu \neq \bot \\ &= \bot \text{ sinon} \end{aligned} \tag{6}$$

▸ **EXERCISE 4.14**
Les constructions $\texttt{let}\ x = M\ \texttt{in}\ N$ et $(\texttt{fun}\ x \rightarrow N)M$ sont-elles équivalentes ? On dira que deux expressions M et N sont *équivalentes* si et seulement si $[\![M]\!]_{\text{caml}}\,\rho\mu = [\![N]\!]_{\text{caml}}\,\rho\mu$ pour tout $\rho \in Env$ et $\mu \in Mem$.

Abordons maintenant les références, ce pour quoi nous avons introduit les mémoires. Pour ceci :

Définition 5 (Stratégie d'allocation mémoire) *Une* stratégie d'allocation mémoire *est une fonction $alloc$ de l'ensemble $\mathbb{P}_{fin}(Addr)$ des ensembles finis d'adresses vers $Addr_\bot$, telle que $alloc(S) \notin S$ pour toute partie finie S de $Addr$.*

Fixons-nous dans la suite une stratégie d'allocation mémoire $alloc$.
En notant ϵ le n-uplet vide (dans $Val^* \subseteq Val$) :

$$\begin{aligned} [\![\texttt{ref}\ M]\!]_{\text{caml}}\,\rho\mu &= (\mu'[a \mapsto V], a) \\ &\quad \text{où } (\mu', V) = [\![M]\!]_{\text{caml}}\,\rho\mu \neq \bot, a = alloc(\text{dom}\ \mu') \neq \bot \\ &= \bot \qquad \text{sinon} \end{aligned} \tag{7}$$

$$[\![!M]\!]_{\text{caml}}\,\rho\mu = (\mu', \mu'(a)) \text{ si } a \in \text{dom}\ \mu',\ \text{où } (\mu', a) = [\![M]\!]_{\text{caml}}\,\rho\mu \neq \bot \tag{8}$$

$$
\begin{aligned}
& = \bot \qquad \text{sinon} \\
[\![M{:=}N]\!]_{\text{caml}}\, \rho\mu & = (\mu''[a \mapsto V], \epsilon) \text{ si } a \in \text{dom } \mu'' \qquad (9)\\
& \quad \text{où } (\mu', a) = [\![M]\!]_{\text{caml}}\, \rho\mu \neq \bot,\ (\mu'', V) = [\![N]\!]_{\text{caml}}\, \rho\mu' \neq \bot \\
& = \bot \text{ sinon}
\end{aligned}
$$

L'effet de `ref` M est donc de calculer M, puis d'allouer une adresse non encore utilisée a, c'est-à-dire hors du domaine de le mémoire courante μ' (si c'est possible, c'est-à-dire si la stratégie retourne autre chose que $\bot$), et met à jour la mémoire de sorte que l'on trouve la valeur V de M à l'adresse a.

Les opérations sur les n-uplets sont maintenant faciles. Notons $V_1 \cdot \ldots \cdot V_n$ le n-uplet (le mot) formé des composantes $V_1, \ldots, V_n$. Nous n'utiliserons pas la notation $(V_1, \ldots, V_n)$, qui pourrait prêter à confusion avec la syntaxe $(M_1, \ldots, M_n)$ des n-uplets dans le langage mini-Caml.

$$
\begin{aligned}
[\![(M_1, \ldots, M_n)]\!]_{\text{caml}}\, \rho\mu & = (\mu_n, V_1 \cdot \ldots \cdot V_n) \qquad (10)\\
& \quad \text{où } \mu_0 = \mu, \text{ et pour tout } i,\ 1 \leq i \leq n, \\
& \qquad (\mu_i, V_i) = [\![M_i]\!]_{\text{caml}}\, \rho\mu_{i-1} \neq \bot \\
& = \bot \quad \text{sinon} \\
[\![\texttt{proj}_i M]\!]_{\text{caml}}\, \rho\mu & = (\mu', V_i) \qquad (11)\\
& \quad \text{si } (\mu', V_1 \cdot \ldots \cdot V_i \cdot \ldots) = [\![M]\!]_{\text{caml}}\, \rho\mu \neq \bot \\
& = \bot \quad \text{sinon}
\end{aligned}
$$

Rappelons que $\mathbb{B} = \{0, 1\}$. La sémantique des autres constructions essentielles de mini-Caml, la séquence et la conditionnelle, est immédiate elle aussi :

$$
\begin{aligned}
[\![M; N]\!]_{\text{caml}}\, \rho\mu & = [\![N]\!]_{\text{caml}}\, \rho\mu' \qquad (12)\\
& \quad \text{où } (\mu', V) = [\![M]\!]_{\text{caml}}\, \rho\mu \neq \bot \\
& = \bot \quad \text{sinon} \\
[\![\texttt{if } M \texttt{ then } N \texttt{ else } P]\!]_{\text{caml}}\, \rho\mu & = [\![N]\!]_{\text{caml}}\, \rho\mu' \qquad (13)\\
& \quad \text{où } (\mu', 1) = [\![M]\!]_{\text{caml}}\, \rho\mu \neq \bot \\
& = [\![P]\!]_{\text{caml}}\, \rho\mu'' \\
& \quad \text{où } (\mu'', 0) = [\![M]\!]_{\text{caml}}\, \rho\mu \neq \bot \\
& =_{def} \bot \quad \text{sinon}
\end{aligned}
$$

Nous ne donnerons pas la sémantique des autres instructions, qui se définissent au cas par cas, or nous n'avons pas défini précisément de quelles autres instructions nous disposons en mini-Caml (voir les trois petits points dans la ligne des opérations arithmétiques en figure 4). Les constructions essentielles ont été vues ci-dessus, de toute faç on.

En supposant que toutes les constructions de mini-Caml ont été décrites (c'est-à-dire en ignorant les constructions arithmétiques, pas souci de simplicité), on peut montrer :

Théorème 13 (Bonne définition, continuité) *Les équations (3)–(13) sont une définition valide de la fonction* $[\![_]\!]_{\text{caml}} : \mathbf{Caml} \times Env \times Mem \to (Mem \times Val)_\bot$. *De plus,* $[\![M]\!]_{\text{caml}}\, \rho\mu$ *est continue en* $\rho \in Env$ *et* $\mu \in Mem$.

Démonstration. Les deux résultats, bonne définition et continuité, se démontrent simultanément, par récurrence structurelle sur M. À noter qu'on a besoin de la continuité de $[\![M]\!]_{\text{caml}}$ ne serait-ce que pour montrer que le côté droit de l'équation (5) définissant la sémantique de $\texttt{fun}\ x \to M$ est bien défini, sans parler de continuité. □

▸ **EXERCISE 4.15**
Montrer qu'une fonction f de $X \times Y$ dans Z, où X, Y, Z sont des cpo, est continue si et seulement si elle est continue en chaque argument séparément. Autrement dit, f est continue de $X \times Y$ vers Z si et seulement si, pour tout $x \in X$, la fonction $y \mapsto f(x, y)$ est continue de Y vers Z, et pour tout $y \in Y$, la fonction $x \mapsto f(x, y)$ est continue de X vers Z. (On rappelle que ce n'est pas le cas pour les fonctions d'un produit de deux espaces topologiques quelconques dans un autre.)

▸ **EXERCISE 4.16**
Démontrez formellement le théorème 13. On pourra s'aider de l'exercice 4.15 pour le cas de l'équation (5).

4.1.3 Sémantique dénotationnelle des programmes

Nous avons donné la sémantique des expressions, il reste à donner celle des programmes. Les programmes mini-Caml sont juste des listes de définitions, récursives ($\texttt{letrec}\ f = \texttt{fun}\ x \to M;;$) ou non ($\texttt{let}\ x = M;;$).

La sémantique d'une telle définition d sera une donnée $[\![d]\!]^{\text{prog}}_{\text{caml}}\,\rho\mu$ dans $(Env \times Mem)_\perp$. En effet, soit la définition échoue (n'a aucun sens), et $[\![d]\!]^{\text{prog}}_{\text{caml}}\,\rho\mu$ vaudra $\perp$, par exemple, si M ne termine pas dans $\texttt{let}\ x = M;;$, soit la définition retourne un nouvel environnement ρ, enrichi de la définition du symbole f ou x, ainsi qu'une nouvelle mémoire au cas où le calcul l'aurait modifiée.

Posons $Fun = [Mem \times Val \to (Mem \times Val)_\perp]$. On rappelle que Fun est un cpo pointé (exercice 4.2). Par le corollaire 3, toute fonction continue F de Fun dans Fun a un plus petit point fixe, notons-le $lfp(F)$ ("least fixed point"). La définition suivante est donc légitime :

$$[\![\texttt{letrec}\ f = \texttt{fun}\ x \to M;;]\!]^{\text{prog}}_{\text{caml}}\,\rho\mu \quad = \quad (\rho[f \mapsto lfp(F)], \mu) \tag{14}$$

où $F : Fun \to Fun$ est la fonction qui à $\phi \in Fun$ associe l'unique ϕ' tel que $(\mu, \phi') = [\![\texttt{fun}\ x \to M;;]\!]_{\text{caml}}\,(\rho[f \mapsto \phi])\mu$: ϕ' est par construction un élément de Fun. De plus, F est continue par l'exercice 4.16.

Le cas du $\texttt{let}$ est plus simple, mais permet de changer la mémoire :

$$\begin{aligned}[\![\texttt{let}\ x = M;;]\!]^{\text{prog}}_{\text{caml}}\,\rho\mu \quad &= \quad (\rho[x \mapsto V], \mu') \\ &\qquad \text{où } (\mu', V) = [\![M]\!]_{\text{caml}}\,\rho\mu \neq \perp \\ &= \quad \perp \quad \text{sinon}\end{aligned} \tag{15}$$

Finalement, un programme étant une séquence de telles définitions, on pose :

$$[\![\epsilon]\!]^{\text{prog}}_{\text{caml}}\,\rho\mu \quad = \quad (\rho, \mu) \tag{16}$$

$$\begin{aligned}
[\![d_1 \; d_2]\!]^{\text{prog}}_{\text{caml}} \, \rho\mu &= [\![d_2]\!]_{\text{caml}} \, \rho'\mu' \\
&\quad \text{où } (\rho', \mu') = [\![d_1]\!]_{\text{caml}} \, \rho\mu \neq \bot \\
&= \bot \qquad \text{sinon}
\end{aligned} \tag{17}$$

5 Leç on 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 faç on 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* $\text{fv}(M)$ *des* variables libres *d'une expression mini-Caml M est défini par récurrence structurelle sur M par :*

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

▸ **EXERCISE 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 $x \in \text{fv}(M)$, alors $[\![M]\!]_{\text{caml}}\, \rho\mu = [\![M]\!]_{\text{caml}}\, \rho'\mu$.

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* $\text{fv}(M) \setminus \{x\} \subseteq \text{dom}\ \varrho$. *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* $\varrho \in Var \rightarrow_{fin} Val'$ *est un environnement fini tel que* $\text{fv}(M) \setminus \{f, x\} \subseteq \text{dom } \varrho$*. 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 :

▸ **EXERCISE 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 :

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

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

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

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.

▸ **EXERCISE 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 reç oit 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) :

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

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'.

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

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

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

▸ **EXERCISE 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) :

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

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 :

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

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*.

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

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 leç on 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

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

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 remplaç ant 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 $\mathtt{a}_1$, d'un environnement fini envoyant x vers 2 et m vers $\mathtt{a}_1$, et que l'on évalue l'expression $\mathtt{let}\ y = !m\ \mathtt{in}\ (m{:=}x; y)$:

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

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 $m \in \mathrm{dom}\ [x \mapsto 2, m \mapsto \mathtt{a}_1]$ en haut de l'instance la plus à gauche de (Var).

Le fait que les dérivations soient des arbres finis, ou de faç onéquivalente 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.) □

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 $\texttt{fix}\langle\texttt{boucle}, \texttt{x}, \texttt{if b then (e; boucle x) else ()}, \varrho_0\rangle$, 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 $\varrho_1[\texttt{x} \mapsto \epsilon], \mu \vdash \texttt{b} \Rightarrow \mu', 0$, la seule dérivation d'un jugement de la forme $\varrho_1, \mu \vdash \texttt{boucle}() \Rightarrow \ldots$ est :

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

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 $\varrho_0[\mathtt{boucle} \mapsto \mathtt{fix}\langle \mathtt{boucle}, \mathtt{x}, \mathtt{if\ b\ then\ (e;\ boucle\ x)\ else\ ()}, \varrho_0\rangle][\mathtt{x} \mapsto \epsilon]$. Notons que cette expression se simplifie en $\varrho_2 = \varrho_1[\mathtt{x} \mapsto \epsilon]$.

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

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

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

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

et

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

(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 $\varrho_1, \mu \vdash \mathtt{boucle}() \Rightarrow \ldots$ 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 faç on, 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*.

▸ **EXERCISE 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 :

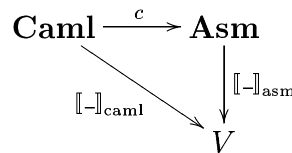
```
let x=2;;
let f = fun y -> x+y;;
let g = fun x -> f x;;
g 3;;
```

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`.

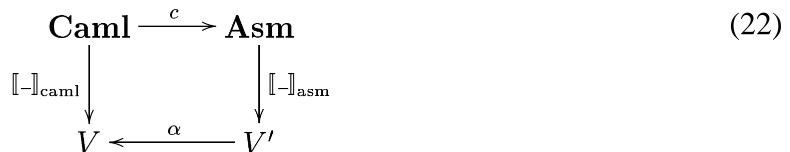
▸ **EXERCISE 5.6**
En utilisant l'exercice 5.5, montrez formellement l'équivalence entre la définition de `while (b) e` comme une abréviation de `boucle ()`, dans tout environnement liant `boucle` à la clôture récursive `fix`$\langle$`boucle`, `x`, `if b then (e; boucle x) else ()`, $\varrho_0\rangle$; 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 faç on abstraite un diagrame de la forme



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 :



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.

▸ **EXERCISE 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 faç on 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

$$\begin{array}{ccc} \mathbf{Caml} & \xrightarrow{c} & \mathbf{Caml} \\ \downarrow{\scriptstyle [\![_]\!]_{\mathrm{caml}}} & & \downarrow{\scriptstyle [\![_]\!]_{\mathrm{caml}'}} \\ V & \xleftarrow{\alpha} & V' \end{array} \qquad (23)$$

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 :

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

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* $\phi(\mu, V) = [\![M]\!]_{\mathrm{caml}}(\rho[x \mapsto V])\mu$*, 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 *;)*
- $\alpha_0(\mathtt{fix}\langle f, x, M, \varrho\rangle) = lfp(F)$*, 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* $(\mu, \phi') = [\![\mathtt{fun}\ x \to M;;]\!]_{\mathrm{caml}}(\rho[f \mapsto \phi])\mu$*, 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 $\mathrm{fv}(M) \setminus \{x\}$, resp. $\mathrm{fv}(M) \setminus \{f, x\}$, 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 $[\![_]\!]_{\mathrm{caml}}$ 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 $[\![_]\!]_{\text{caml}}$ 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* $[\![_]\!]_{\text{caml}'}$ *est* correcte *par rapport à la sémantique* $[\![_]\!]_{\text{caml}}$*, 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* $\text{fv}(M) \subseteq \text{dom } \varrho$*, et tout environnement* ρ *tel que* $\rho(y) = \alpha_0(\varrho(y))$ *pour tout variable* $y \in \text{fv}(M)$*,*
- *pour toute mémoire concrète* μ*, pour toute mémoire* $\tilde{\mu}$ *telles que* $\text{dom } \mu \subseteq \text{dom } \tilde{\mu}$ *et* $\tilde{\mu}(a) = \alpha_0(\mu(a))$ *pour tout* $a \in \text{dom } \mu$*,*

alors :

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

Bien sûr, ou bien $[\![c(M)]\!]_{\text{caml}'}\, \varrho\mu = \emptyset$ ou bien $[\![c(M)]\!]_{\text{caml}'}\, \varrho\mu$ 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 $[\![c(M)]\!]_{\text{caml}'}\, \varrho\mu$ 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 $[\![M]\!]_{\text{caml}}\, \rho\mu$ est de la forme $(\tilde{\mu}', V) \neq \perp$, avec $V = \alpha_0(v)$, $\text{dom } \mu' \subseteq \text{dom } \tilde{\mu}'$ et $\tilde{\mu}'(a) = \alpha_0(\mu'(a))$ pour tout $a \in \text{dom } \mu'$. 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 $[\![M]\!]_{\text{caml}}\, \rho\tilde{\mu} \neq \perp$, 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* id : $\mathbf{Caml} \to \mathbf{Caml}$ *: 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.

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

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

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. □

A Guide de référence rapide de l'assembleur Pentium

Les courageux pourront consulter `http://www.intel.com/design/intarch/techinfo/pentium/instsum.htm` pour une description complète du jeu d'instruction du Pentium. Nous n'aurons besoin dans le cours que de ce qui est décrit dans cette annexe.

Les registres : `%eax %ebx %ecx %edx %esi %edi %ebp %esp`.
Ils contiennent tous un entier de 32 bits (4 octets), qui peut aussi être vu comme une adresse. Le registre `%esp` est spécial, et pointe sur le sommet de pile ; il est modifié par les instructions `pushl`, `popl`, `call`, `ret` notamment.

Il y a aussi d'autres registres que l'on ne peut pas manipuler directement. (L'instruction `info registers` sous `gdb` ou `ddd` vous les montrera.) Le plus important est `%eip`, le *compteur de programme* : il contient en permanence l'adresse de la prochaine instruction à exécuter.

- `addl` ⟨source⟩, ⟨dest⟩ . . . . ⟨dest⟩= ⟨dest⟩+ ⟨source⟩ (addition)
  Ex : `addl $1, %eax` ajoute 1 au registre `%eax`.
  Ex : `addl $4, %esp` dépile un élément de 4 octets de la pile.
  Ex : `addl %eax, (%ebx, %edi, 4)` ajoute le contenu de `%eax` à la case mémoire à l'adresse `%ebx` $+ 4 *$ `%edi`. (Imaginez que `%ebx` est l'adresse de début d'un tableau `a`, `%edi` est un index `i`, ceci stocke `%eax` dans `a[i]`.)

- `andl` ⟨source⟩, ⟨dest⟩ . . . . ⟨dest⟩= ⟨dest⟩& ⟨source⟩ (et bit à bit)
- `call` ⟨dest⟩ . . . . appel de procédure à l'adresse ⟨dest⟩
  Équivalent à `pushl $`a, où a est l'adresse juste après l'instruction `call` (l'adresse *de retour*), suivi de `jmp` ⟨dest⟩.
  Ex : `call printf` appelle la fonction `printf`.
  Ex : `call *%eax` (appel indirect) appelle la fonction dont l'adresse est dans le registre `%eax`. Noter qu'il y a une irrégularité dans la syntaxe, on écrit `call *%eax` et non `call (%eax)`.

- `cltd` . . . . conversion 32 bits → 64 bits
  Convertit le nombre 32 bits dans `%eax` en un nombre sur 64 bits stocké à cheval entre `%edx` et `%eax`.
  Note : `%eax` n'est pas modifié ; `%edx` est mis à 0 si `%eax` est positif ou nul, à -1 sinon.
  À utiliser notamment avant l'instruction `idivl`.

- `cmpl` ⟨source⟩, ⟨dest⟩ . . . . comparaison
  Compare les valeurs de ⟨source⟩ et ⟨dest⟩. Utile juste avant un saut conditionnel (`je`, `jge`, etc.). À noter que la comparaison est faite dans le sens inverse de celui qu'on attendrait. Par exemple, `cmp` ⟨source⟩, ⟨dest⟩ suivi d'un `jge` ('jump if greater than or equal to"), va effectuer le saut si ⟨dest⟩≥⟨source⟩ : on compare ⟨dest⟩ à ⟨source⟩, et non le contraire.

- `idivl` ⟨dest⟩ . . . . division entière et reste
  Divise le nombre 64 bits stocké en `%edx` et `%eax` (cf. `cltd`) par le nombre 32 bits ⟨dest⟩. Re-

tourne le quotient en `%eax`, le reste en `%edx`.

- `imull` ⟨source⟩, ⟨dest⟩ . multiplie ⟨dest⟩ par ⟨source⟩, résultat dans ⟨dest⟩
- `jmp` ⟨dest⟩ .... saut inconditionnel : `%eip`=⟨dest⟩
- `je` ⟨dest⟩ .... saut conditionnel
  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩=⟨source⟩, continue avec le fbt normal du programme sinon.

- `jg` ⟨dest⟩ .... saut conditionnel
  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩>⟨source⟩, continue avec le fbt normal du programme sinon.

- `jge` ⟨dest⟩ .... saut conditionnel
  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩≥⟨source⟩, continue avec le fbt normal du programme sinon.

- `jl` ⟨dest⟩ .... saut conditionnel
  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩<⟨source⟩, continue avec le fbt normal du programme sinon.

- `jle` ⟨dest⟩ .... saut conditionnel
  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩≤⟨source⟩, continue avec le fbt normal du programme sinon.

- `leal` ⟨source⟩, ⟨dest⟩ .... chargement d'adresse effective
  Au lieu de charger le contenu de ⟨source⟩ dans ⟨dest⟩, charge l'*adresse* de ⟨source⟩.
  Équivalent C : ⟨dest⟩=&⟨source⟩.

- `movl` ⟨source⟩, ⟨dest⟩ .... transfert
  Met le contenu de ⟨source⟩ dans ⟨dest⟩. Équivalent C : ⟨dest⟩=⟨source⟩.
  Ex : `movl %esp, %ebp` sauvegarde le pointeur de pile `%esp` dans le registre `%ebp`.
  Ex : `movl %eax, 12(%ebp)` stocke le contenu de `%eax` dans les quatre octets commençant à `%ebp` + 12.
  Ex : `movl (%ebx, %edi, 4), %eax` lit le contenu de la case mémoire à l'adresse `%ebx` + 4 ∗ `%edi`, et le met dans `%eax`. (Imaginez que `%ebx` est l'adresse de début d'un tableau `a`, `%edi` est un index `i`, ceci stocke `a[i]` dans `%eax`.)

- `negl` ⟨dest⟩ .... ⟨dest⟩=-⟨dest⟩(opposé)
- `notl` ⟨dest⟩ .... ⟨dest⟩=~⟨dest⟩(non bit à bit)
- `orl` ⟨source⟩, ⟨dest⟩ .... ⟨dest⟩= ⟨dest⟩ | ⟨source⟩ (ou bit à bit)
- `popl` ⟨dest⟩ .... dépilement

Dépile un entier 32 bits de la pile et le stocke en ⟨dest⟩.
Équivalent à `movl (%esp),`⟨dest⟩ suivi de `addl $4, %esp`.
Ex : `popl %ebp` récupère une ancienne valeur de `%ebp` sauvegardée sur la pile, typiquement, par `pushl`.

– `pushl` ⟨source⟩ . . . . empilement
Empile l'entier 32 bits ⟨source⟩ au sommet de la pile.
Équivalent à `movl` ⟨source⟩`, -4(%esp)` suivi de `subl $4, %esp`.
Ex : `pushl %ebp` sauvegarde la valeur de `%ebp`, qui sera rechargée plus tard par `popl`.
Ex : `pushl` ⟨source⟩ permet aussi d'empiler les arguments successifs d'une fonction. (Note : pour appeler une fonction C comme `printf` par exemple, il faut empiler les arguments en commençant par celui de droite.)

– `ret` . . . . retour de procédure
Dépile une adresse de retour a, et s'y branche. Lorsque la pile est remise dans l'état à l'entrée d'une procédure f, ceci a pour effet de retourner de f et de continuer l'exécution de la procédure appelante.
Équivalent à `popl %eip`... si cette instruction existait (il n'y a pas de mode d'adressage permettant de manipuler `%eip` directement).

– `subl` ⟨source⟩, ⟨dest⟩ . . . . ⟨dest⟩= ⟨dest⟩− ⟨source⟩(soustraction)
Ex : `subl $1, %eax` retire 1 du registre `%eax`.
Ex : `subl $4, %esp` alloue de la place pour un nouvel élément de 4 octets dans la pile.

– `xorl` ⟨source⟩, ⟨dest⟩ . . . . . ⟨dest⟩= ⟨dest⟩^ ⟨source⟩ (ou exclusif bit à bit)