3 Leçon 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

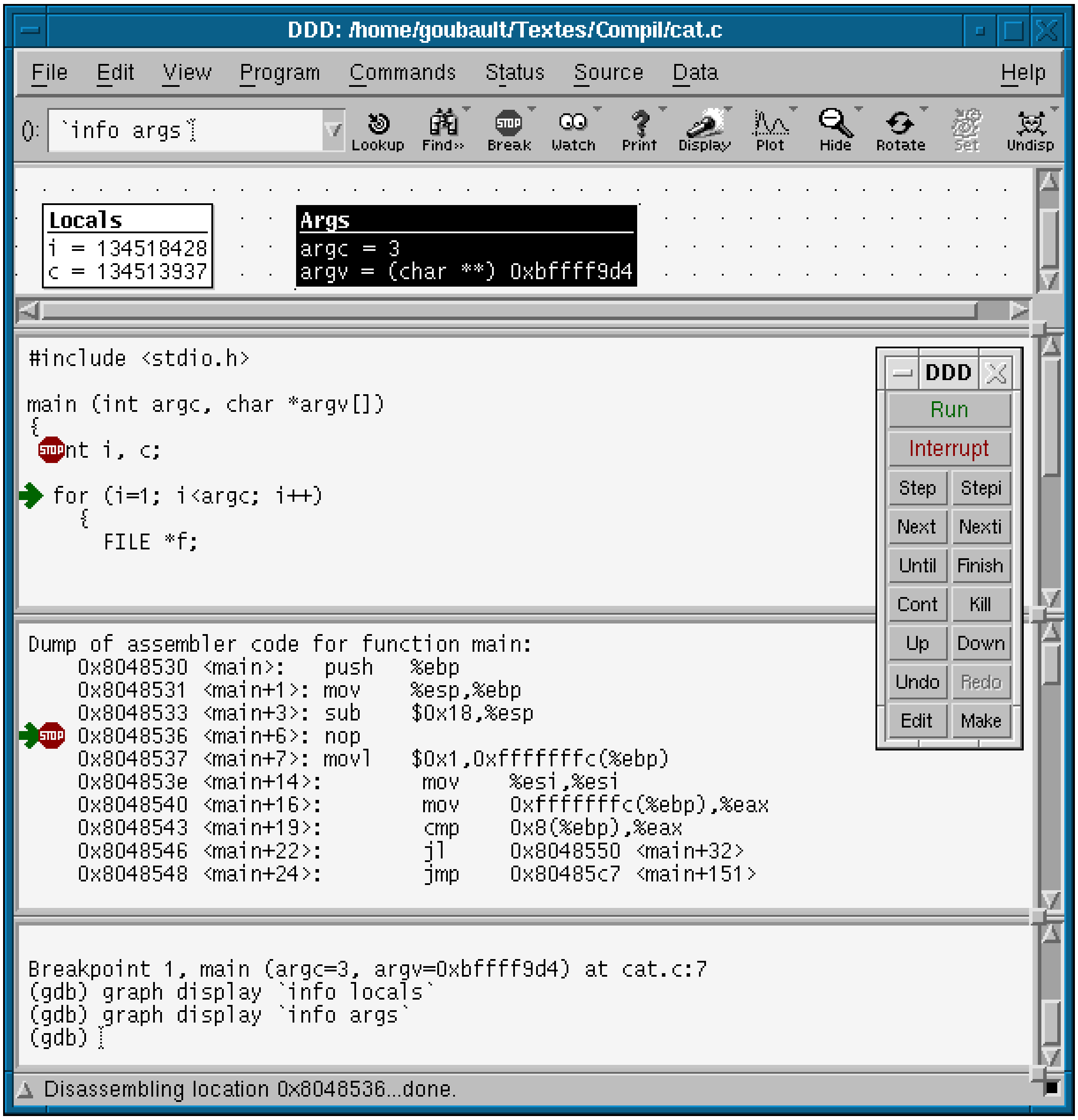
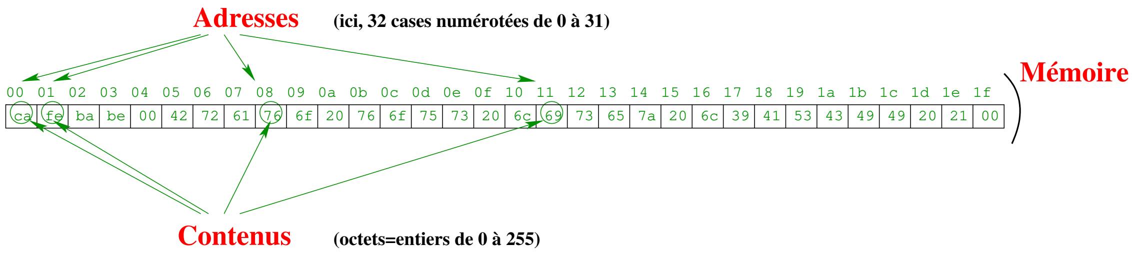


FIG. 5 – Une session sous ddd, montrant ensemble code assembleur et code C



Octets :

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

Electronique :

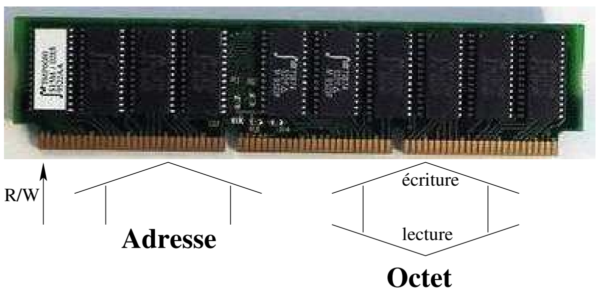


FIG. 6 – La mémoire des ordinateurs

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.

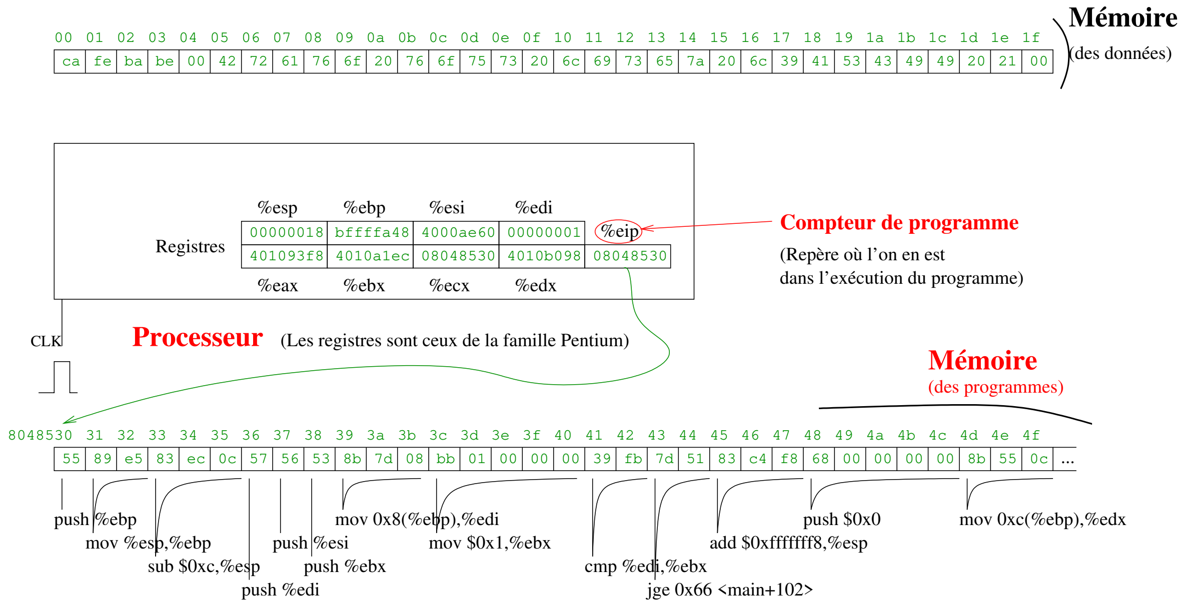


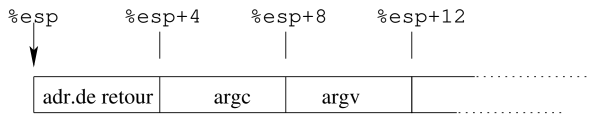
FIG. 8 – Le processeur et la mémoire

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

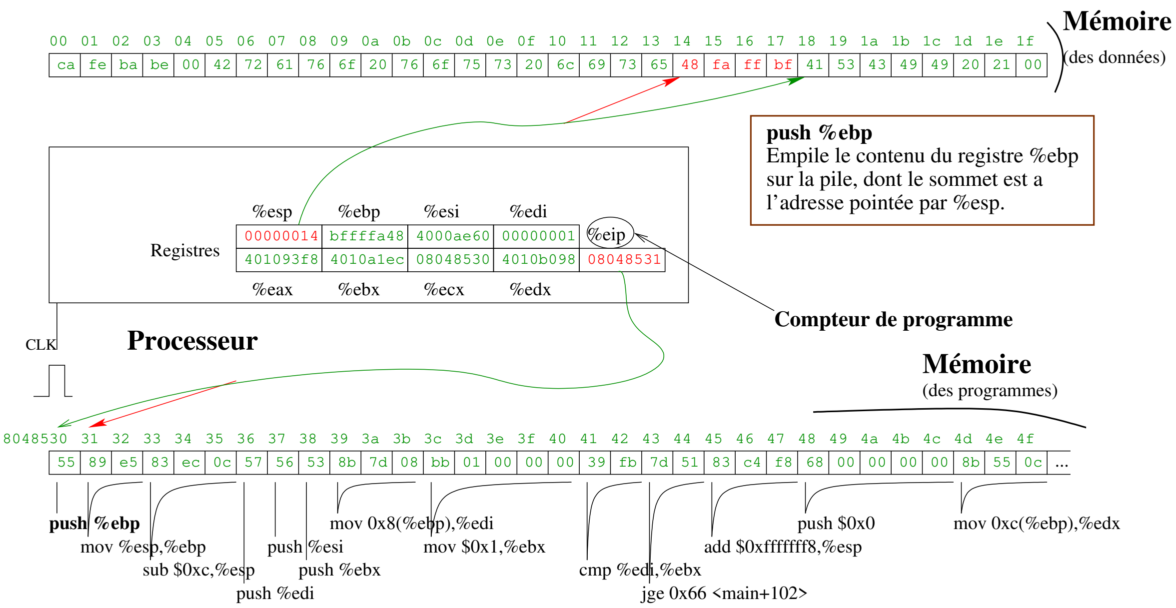


FIG. 9 – Exécution de la première instruction

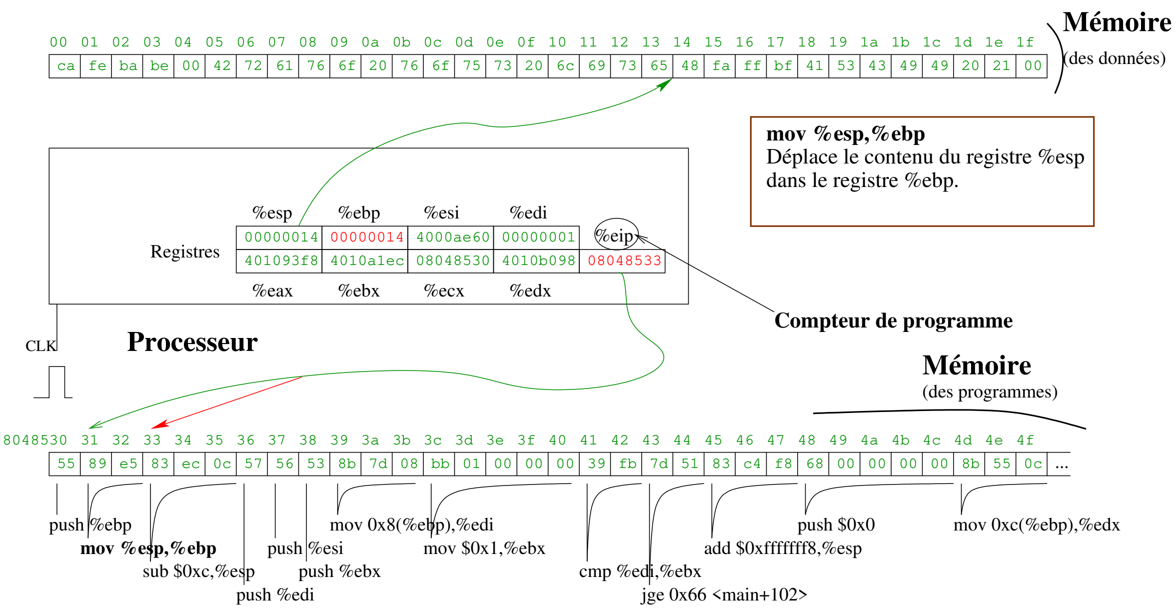
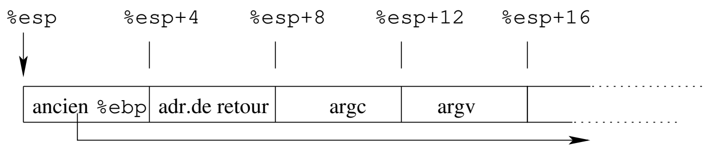


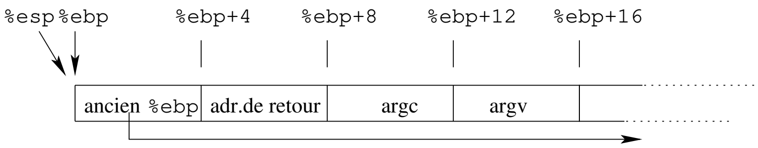
FIG. 10 – Exécution de la deuxième instruction

(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.

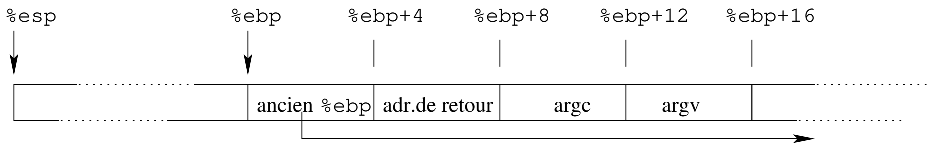
Après l'instruction `pushl %ebp`, la pile ressemble à :



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 :

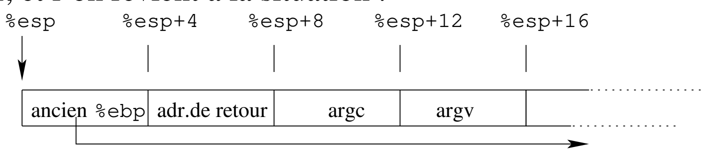


Concrètement, la situation est comme montrée sur la figure 11. Au lieu de soustraire 12 octets, on aurait pu en ajouter -12, ce qui se serait fait avec l'instruction `addl $-12, %esp`, ou de façon équivalente, `addl $0xfffffff4, %esp`. (Donc, que fait l'instruction `addl $0xfffffff8, %esp` un peu plus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

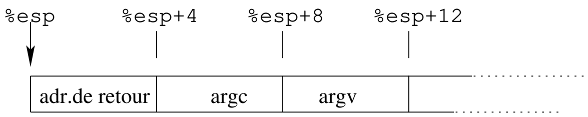
L'intérêt de l'allocation des 12 octets avant `%ebp` est de permettre de réserver de la place pour les trois variables `i`, `c` et `f` de la fonction `main`, typiquement.

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 :



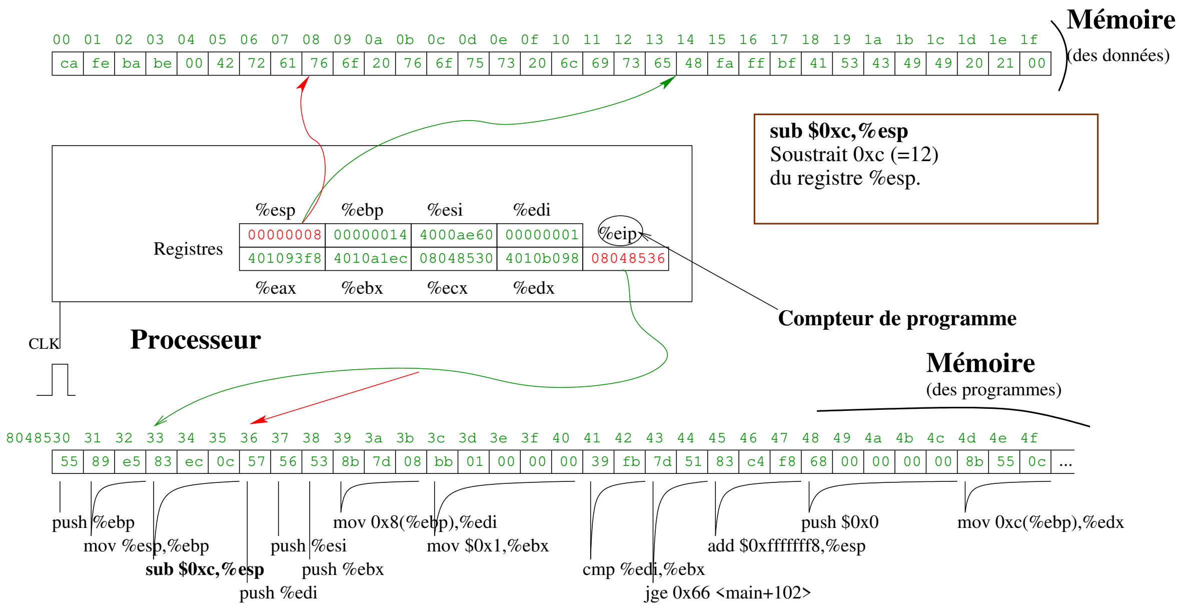


FIG. 11 – Exécution de la troisième instruction

– `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).

▸ **EXERCICE 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.)

▸ **EXERCICE 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.

▸ **EXERCICE 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 ?

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

▸ **EXERCICE 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 :

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

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.)

▸ **EXERCICE 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.

▸ **EXERCICE 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 $\langle\text{dest}\rangle$ 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 $\langle\text{dest}\rangle$, 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 $regs(r')$.

▸ **EXERCICE 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 !

▸ **EXERCICE 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 commençant à l'adresse pc. Ceci détermine en particulier de façon 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 $\langle$source$\rangle$=$\langle$dest$\rangle$. Ceci permettra notamment de tester l'égalité de $\langle$source$\rangle$ avec $\langle$dest$\rangle$. Le flag `CF` permet, lui, de tester si $\langle$dest$\rangle$<$\langle$source$\rangle$ (en non signé). L'utilisation des autres flags est un peu plus compliquée, et est laissée en exercice.

▸ **EXERCICE 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 $\langle$dest$\rangle$<$\langle$source$\rangle$ 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 :

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

▸ **EXERCICE 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 :

$$\begin{tikzcd} \mathbf{Caml} \arrow[r, "c"] \arrow[dr, "{[\![\_]\!]_{\mathrm{caml}}}"'] & \mathbf{Asm} \arrow[d, "{[\![\_]\!]_{\mathrm{asm}}}"] \\ & Val \end{tikzcd} \tag{1}$$

(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 `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 `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. $\square$

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.

▸ **EXERCICE 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.

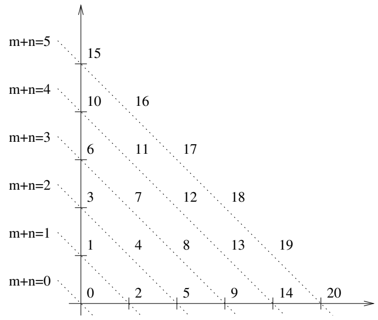


FIG. 12 – Codage des paires d'entiers

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 $e_{1690} = \{1, 3, 4, 7, 9, 10\}$.

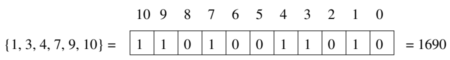


FIG. 13 – Codage des ensembles finis

▸ **EXERCICE 3.13**
Trouver une formule pour $\mathtt{proj}_1 : \langle m, n\rangle \mapsto m$ et pour $\mathtt{proj}_2 : \langle m, n\rangle \mapsto n$.

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 français, 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 $\mathbb{P}\omega$:

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

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

▸ **EXERCICE 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").

▸ **EXERCICE 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 $([0,1], \leq)$.

▸ **EXERCICE 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 ?