Programmation 1: sémantique, leçon 1

Jean Goubault-Larrecq
ENS Cachan
goubault@lsv.ens-cachan.fr

14 décembre 2020

Résumé

Ceci est la version 3 du 14 décembre 2020. La version 2 datait du 12 novembre 2014, et avait bénéficié de corrections de bugs dans la version 1, dénichés par Alexis Ghyselen et Glen Mével. La version 1 datait du 18 novembre 2013.

1 Sémantique

Que fait, ou plutôt que devrait faire, un programme π ? Connaissant le langage dans lequel π est écrit, on peut le compiler, l'exécuter, et inférer ce que π fait à partir d'exemples.

Il est plus satisfaisant d'un point de vue scientifique de disposer d'une définition mathématique $[\![\pi]\!]$ de *ce que calcule* π (la *sémantique* de π). Avec cette définition, on pourra raisonner, prouver des théorèmes — notamment des théorèmes de *correction*, exprimant que π calcule bien une donnée ayant des propriétés souhaitées —, voire définir des logiques permettant de raisonner sur les programmes, comme la logique de Hoare.

Il existe de nombreux styles de sémantiques. Je vais vous en présenter (au moins) deux : les sémantiques dites *opérationnelles* (et plus précisément, à petits pas) en section 1.2 et les sémantiques *dénotationnelles* (dans le prochain poly). Je présenterai les deux sur une version terriblement expurgée d'un langage impératif, appelé IMP ; le but n'est pas pour l'instant de donner un sémantique à un vrai langage, mais à s'entraîner à travailler avec une sémantique.

Je vous décrirai ensuite, en section 1.6, une sémantique possible pour le langage C−− dont vous avez à réaliser un compilateur en projet de programmation (partie 1).

Nous verrons des sémantiques d'autres langages, notamment fonctionnels, dans les leçons suivantes.

1.1 IMP

Considérons le langage IMP dont la syntaxe est la suivante :

Commandes			Expressions		
c ::=	$x := e$	affectation	e ::=	x	variables
|	`skip`	ne rien faire	|	$\dot{n}$	constante entière ($n \in \mathbb{Z}$)
|	$c_1; c_2$	séquence	|	$e \dot{+} e$	addition
|	`if` e `then` c_1 `else` c_2	conditionnelle	|	$\dot{-} e$	opposé
|	`while` e `do` c	boucle while			

On peut difficilement faire plus simple. On pourrait inclure de nombreuses autres constructions, tant de commandes que d'expressions, mais celles-ci nous suffiront pour l'instant. (De plus, le langage est déjà Turing-complet !)

Il faudra bien faire attention à distinguer la *syntaxe* (ci-dessus) de la *sémantique* (les valeurs calculées). Pour enfoncer le clou, j'ai décidé de noter $\dot{+}$, resp. $\dot{-}$, certains symboles dont la *sémantique* sera l'addition $+$, respectivement l'opposé $-$. Les symboles $\dot{+}$ et $\dot{-}$ ne sont que des lettres, et *rien d'autre*.

Commençons par donner une sémantique des expressions e. Idéalement, on aimerait pouvoir définir une valeur $[\![e]\!]$, en disant par exemple que $[\![\dot{3}\dot{2}]\!] = 32$, $[\![e_1 \dot{+} e_2]\!] = [\![e_1]\!] + [\![e_2]\!]$, etc. Mais il nous faudra aussi donner la valeur des variables. Pour ceci, nous paramétrons la sémantique par un *environnement*, c'est-à-dire une fonction (totale) ρ qui à chaque variable associe sa valeur.

Nous supposerons que IMP calcule sur des entiers dans $\mathbb{Z}$, l'ensemble Env des environnements sera donc l'ensemble $Var \to \mathbb{Z}$ de toutes les fonctions de l'ensemble Var des variables vers $\mathbb{Z}$. (Nous ignorons donc le fait qu'un programme réel calculerait plutôt sur des entiers machines.)

On pose ainsi :

$$\begin{aligned}
[\![x]\!]\,\rho &= \rho(x) \\
[\![\dot{n}]\!]\,\rho &= n \\
[\![e_1 \dot{+} e_2]\!]\,\rho &= [\![e_1]\!]\,\rho + [\![e_2]\!]\,\rho \\
[\![\dot{-} e]\!]\,\rho &= -\,[\![e]\!]\,\rho.
\end{aligned}$$

C'est un exemple (très simple, et quasi-creux), de sémantique *dénotationnelle* : une fonction sémantique $[\![\_]\!]\,\_$ qui à chaque couple expression, environnement associe une *valeur*.

Nous aurons plus de difficulté à définir une sémantique opérationnelle des commandes. On serait tenté de définir $[\![c]\!]\,\rho$, pour chaque commande c et environnement ρ (avant l'exécution de la commande), comme valant l'environnement à la fin de l'exécution de la commande, par :

$$\begin{aligned}
[\![x := e]\!]\,\rho &= \rho[x \mapsto [\![e]\!]\,\rho] \\
[\![\texttt{skip}]\!]\,\rho &= \rho \\
[\![c_1; c_2]\!]\,\rho &= [\![c_2]\!]\,([\![c_1]\!]\,\rho) \\
[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\,\rho &= \begin{cases} [\![c_1]\!]\,\rho & \text{si } [\![e]\!]\,\rho \neq 0 \\ [\![c_2]\!]\,\rho & \text{si } [\![e]\!]\,\rho = 0 \end{cases}
\end{aligned}$$

et finalement :

$$[\![\texttt{while}\, e\, \texttt{do}\, c]\!]\,\rho \;=\; \begin{cases} \rho & \text{si } [\![e]\!]\,\rho = 0 \\ [\![\texttt{while}\, e\, \texttt{do}\, c]\!]\,([\![c]\!]\,\rho) & \text{si } [\![e]\!]\,\rho \neq 0 \end{cases} \tag{1}$$

La notation $\rho[x \mapsto V]$ est par définition la fonction qui à x associe x à V, et à toute variable $y \neq x$ associe $\rho(y)$. Attention ! $\rho[x \mapsto V]$ n'est pas le résultat d'une quelconque affectation sur ρ, ou modification dans ρ. C'est un nouvel environnement, en général distinct de ρ. Il n'y a pas de notion d'effet de bord en mathématiques.

Dans les règles du `if` et du `while`, on notera que le faux est représenté par 0, et toute valeur non nulle est considérée comme vraie. C'est la convention utilisée en C, notamment.

Cette sémantique est simple (surtout relativement à certaines qui vont suivre), et surtout *compositionnelle* : la sémantique d'une commande est définie à partir de la sémantique de sous-commandes.

Mais l'équation (1) pose un problème : on définit $[\![\texttt{while}\, e\, \texttt{do}\, c]\!]\,\rho$ en fonction de $[\![\texttt{while}\, e\, \texttt{do}\, c]\!]\,\rho'$ (pour l'environnement ρ' obtenue comme résultat de l'évaluation du corps c de la boucle `while`). Ceci n'est donc pas une définition bien fondée.

Pour corriger ce problème, on remarque que la fonction $[\![\texttt{while}\, e\, \texttt{do}\, c]\!]\,\_ : Env \to Env$ doit être un point fixe f de la fonctionnelle $F_{e,c}$: $(Env \to Env) \to (Env \to Env)$ suivante :

$$F_{e,c}(f)(\rho) \;=\; \begin{cases} \rho & \text{si } [\![e]\!]\,\rho = 0 \\ f([\![c]\!]\,\rho) & \text{si } [\![e]\!]\,\rho \neq 0 \end{cases}$$

En supposant l'existence d'une fonction lfp qui prend une telle fonctionnelle et retourne son (ou plutôt un de ses) points fixes, on pourra donc remplacer l'équation absurde (1) par :

$$[\![\texttt{while}\, e\, \texttt{do}\, c]\!]\,\rho \;=\; \mathrm{lfp}(F_{e,c})(\rho). \tag{2}$$

Encore faudra-t-il :

- Trouver un cadre mathématique où l'existence de tels points fixes est assurée. Ce sera la *théorie des domaines*, que nous verrons plus tard. L'espace de toutes les fonctions $Env \to Env$ sera bien trop gros, et nous devrons nous limiter aux fonctions *continues*, dans un certain sens.
- Montrer que cette sémantique calcule bien ce que l'on pense. Le fait que lfp retourne un point fixe parmi plusieurs possibles est troublant. Nous verrons que le bon est le *plus petit* point fixe, ce que nous vérifierons par un théorème d'adéquation entre la sémantique dénotationnelle et une autre sémantique, dite opérationnelle.

Nous ferons tout cela plus tard. Nous nous réservons pour le cas des langages *fonctionnels*, où une sémantique dénotationnelle prendra tout son sens. Le cas de IMP sera alors d'une trivialité navrante.

1.2 IMP : sémantique opérationnelle à petits pas, premier essai

Autant une sémantique dénotationnelle est parfaite pour les expressions (sans effet de bord, sans boucle), autant il est probablement plus simple de passer à un autre style pour décrire le comportement des commandes, des programmes : une sémantique *opérationnelle* d'un programme est juste la description (plus ou moins abstraite) d'une machine qui évalue le programme π.

$$\frac{}{(x := e, \rho) \to \rho[x \mapsto [\![e]\!]\,\rho]}\ (\to{:=}) \qquad \frac{}{(\texttt{skip}, \rho) \to \rho}\ (\to \texttt{skip})$$

$$\frac{(c_1, \rho) \to \rho'}{(c_1; c_2, \rho) \to (c_2, \rho')}\ (\to Seq_{fin}) \qquad \frac{(c_1, \rho) \to (c_1', \rho')}{(c_1; c_2, \rho) \to (c_1'; c_2, \rho')}\ (\to Seq)$$

$$\frac{}{(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \rho) \to (c_1, \rho)}\ (\to \texttt{if}\,1) \quad \text{si } [\![e]\!]\,\rho \neq 0$$

$$\frac{}{(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \rho) \to (c_2, \rho)}\ (\to \texttt{if}\,2) \quad \text{si } [\![e]\!]\,\rho = 0$$

$$\frac{}{(\texttt{while}\, e\ \texttt{do}\ c, \rho) \to (c; \texttt{while}\, e\ \texttt{do}\ c, \rho)}\ (\to \texttt{while}) \quad \text{si } [\![e]\!]\,\rho \neq 0$$

$$\frac{}{(\texttt{while}\, e\ \texttt{do}\ c, \rho) \to \rho}\ (\to \texttt{while}_{fin}) \quad \text{si } [\![e]\!]\,\rho = 0$$

FIGURE 1 – Une sémantique opérationnelle à petits pas de IMP (premier essai)

Il est commode de spécifier cette machine sous forme d'un système de transitions (un automate, mais sur un nombre en général infini d'états), autrement dit sous forme de la donnée d'un ensemble d'états, et d'une relation binaire $\to$ sur l'ensemble des états.

Dans le cas de IMP, on pourra par exemple définir un état comme un couple (ℓ, ρ) où :

— ℓ est le *point de programme* courant (typiquement le numéro de ligne, si chaque commande est écrite sur une ligne) ;

— ρ est un environnement.

Attention ! Ceci est un premier essai, et nous verrons une meilleure sémantique, plus simple et plus claire, en section 1.3.

La notion de point de programme ne fait pas partie de notre syntaxe. On pourrait la rajouter, mais on peut aussi faire d'autres choix. Définissons un *état* de notre sémantique opérationnelle à venir comme :

— soit un couple (c, ρ) où c est une commande et ρ est un environnement (avant l'exécution de c) ;

— soit un environnement ρ (obtenue à terminaison de la commande).

La relation $\to$ sera une relation binaire entre états : $q \to q'$ si et seulement si, intuitivement, dans l'état q, le programme peut arriver en une étape à l'état q'.

Nous allons définir la relation $\to$ par un ensemble de règles, données en figure 1.

La règle $(\to{:=})$ exprime que $x := e$ termine tout de suite, sur l'environnement $\rho[x \mapsto [\![e]\!]\,\rho]$. $[\![e]\!]\,\rho$ est la sémantique de l'expression e, telle que définie en section 1.1. (Nous nous autorisons effectivement à mélanger un peu de sémantique dénotationnelle, pour les expressions, avec notre sémantique opérationnelle.)

On notera que ni la règle $(\to{:=})$ ni la règle $(\to \texttt{skip})$ n'a de prémisse : on peut démontrer que $(x := e, \rho) \to \rho[x \mapsto [\![e]\!]\,\rho]$ *sans condition*. La règle $(\to Seq_{fin})$, qui décrit le comportement de la séquence $c_1; c_2$ (lorsque c_1 retourne en une étape ; sinon c'est la règle $(\to Seq)$ qui s'applique), ne s'applique que si l'on a préalablement démontré $(c_1, \rho) \to \rho'$ — la *prémisse* de la règle. En

général, les prémisses d'une règle sont écrites au-dessus de la barre, et la conclusion en-dessous.

Les règles du `if` et du `while` n'ont aucune prémisse, mais ne peuvent être appliquées que si certaines *conditions de bord* ($[\![e]\!]\,\rho \neq 0$, $[\![e]\!]\,\rho = 0$) sont vérifiées.

Formellement, la relation $\to$ est définie par : $\to$ est *la plus petite* relation entre états telle que, pour chacune des règles de la figure 1, si les prémisses en sont vraies alors la conclusion est vraie aussi.

Ceci peut s'expliquer plus concrètement comme suit.

Une *dérivation* d'un fait $q \to q'$ est un arbre fini étiqueté par des instances des règles de façon cohérente, et dont la racine est $q \to q'$. Plutôt que de définir ceci formellement, donnons un exemple. L'arbre suivant :

$$\dfrac{\dfrac{\dfrac{}{(x := x+1, \rho) \to \rho'}\,(\to :=)}{(x := x+1; y := x, \rho) \to (y := x, \rho')}\,(\to Seq_{fin})}{((x := x+1; y := x); x := 3 + (-y), \rho) \to (y := x; x := 3 + (-y), \rho')}\,(\to Seq)$$

est une dérivation de $((x := x+1; y := x); x := 3 + (-y), \rho) \to (y := x; x := 3 + (-y), \rho')$, où $\rho(x) = 23$, et $\rho' = \rho[x \mapsto 24]$. On notera que :

— La conclusion désirée est tout en bas.

— La dérivation commence, tout en haut, par des règles n'ayant aucune prémisse.

— Chaque barre relie des prémisses (au-dessus de la barre) à la conclusion (en-dessous de la barre) de la règle dont le nom est donnée sur le côté.

Bien sûr, nos arbres sont dégénérés : nos règles n'ayant pas plus d'une prémisse, toutes nos dérivations sont en fait des *suites* d'applications de règles.

Une fois ceci établi, on peut maintenant définir une *trace d'exécution*, qui est une suite d'états $q_0, q_1, \cdots, q_n, \cdots$ (finie ou infinie) telle que $q_0 \to q_1$, $q_1 \to q_2$, etc., sont tous dérivables. On écrira $q_0 \to q_1 \to q_2 \to \cdots \to q_n \to \cdots$ en abrégé.

1.3 La sémantique opérationnelle à petits pas de IMP

La sémantique opérationnelle à petits pas de la section 1.2 souffre finalement d'une complication, dû aux règles de la séquence, et qui impose de produire des dérivations, juste pour trouver la transition à exécuter à l'intérieur d'une séquence.

On peut corriger ce problème facilement en utilisant une sémantique opérationnelle à petits pas qui inclut non seulement la commande à exécuter c mais aussi son *contexte* C. En général, un contexte est un programme avec un trou (souvent noté $\_$), $C[c]$ dénote le programme C où le trou est remplacé par la commande c. Le fait de donner à la fois C et c donne une description du programme total ($C[c]$) et de l'endroit où on en est (la position du trou).

Il n'y a aucune raison de se limiter à cette forme de contextes. Pour marquer la différence, je noterai $c \cdot C$ plutôt que $C[c]$, et le contexte C sera une *liste* finie de commandes à exécuter une fois que c aura terminé. Typiquement, plutôt que d'écrire $C[c]$ où C est le contexte $\_; c_2; c_3; \cdots; c_n$, j'écrirai $c \cdot c_2 \cdot c_3 \cdot \cdots \cdot c_n \cdot \epsilon$ (ϵ étant la liste vide). On a les règles de la figure 2.

$$
\begin{array}{rcll}
(x := e \cdot C, \rho) & \to & (C, \rho[x \mapsto [\![e]\!]\,\rho]) & (3)\\
(\texttt{skip} \cdot C, \rho) & \to & (C, \rho) & (4)\\
(c_1; c_2 \cdot C, \rho) & \to & (c_1 \cdot c_2 \cdot C, \rho) & (5)\\
(\texttt{if}\, e\, \texttt{then}\, c_1\ \texttt{else}\ c_2 \cdot C, \rho) & \to & (c_1 \cdot C, \rho) \quad \text{si } [\![e]\!]\,\rho \neq 0 & (6)\\
(\texttt{if}\, e\, \texttt{then}\, c_1\ \texttt{else}\ c_2 \cdot C, \rho) & \to & (c_2 \cdot C, \rho) \quad \text{si } [\![e]\!]\,\rho = 0 & (7)\\
(\texttt{while}\, e\, \texttt{do}\, c \cdot C, \rho) & \to & (c \cdot \texttt{while}\, e\, \texttt{do}\, c \cdot C, \rho) \quad \text{si } [\![e]\!]\,\rho \neq 0 & (8)\\
(\texttt{while}\, e\, \texttt{do}\, c \cdot C, \rho) & \to & (C, \rho) \quad \text{si } [\![e]\!]\,\rho = 0 & (9)
\end{array}
$$

FIGURE 2 – La sémantique à petits pas de IMP

Cette sémantique est (normalement) plus simple à appréhender que celle de la section 1.2. On pourra comparer les règles pour la séquence ($\to Seq_{fin}$) et ($\to Seq$), avec l'unique nouvelle règle (5), par exemple.

D'autre part, on n'a plus à considérer de système de règle avec prémisses, comme en section 1.2. La relation binaire $\to$ est désormais définie simplement par $(c \cdot C, \rho) \to (c' \cdot C', \rho')$ si et seulement s'il existe une instance d'une règle parmi celles de la figure 2 dont le côté gauche est $(c \cdot C, \rho)$ et le côté droit est $(c' \cdot C', \rho')$. (Pour les gens à l'esprit compliqué, la figure 2 est un système de règles dont aucune n'a de prémisse.)

On notera aussi que les règles du `while` sont bien fondées, contrairement à ce qui se passait dans notre premier essai de sémantique dénotationnelle (section 1.1).

Par exemple, considérons le programme (la commande) suivant :

$$
\underbrace{x := 3; \underbrace{y := 1; \underbrace{\texttt{while}\, x\, \texttt{do}\, \underbrace{(\underbrace{y := y + x; x := x + (-1)}_{c_3})}_{c_2}}_{c_1}}_{c_0}
$$

Voici sa trace d'exécution maximale partant de l'état $(c_0 \cdot \epsilon, \rho_0)$, où ρ_0 est un environnement

arbitraire :

$$
\begin{array}{lll}
(c_0 \cdot \epsilon, \rho_0) & \to & (x := 3 \cdot c_1 \cdot \epsilon, \rho_0) \quad \text{par } (5) \\
& \to & (c_1 \cdot \epsilon, \rho_0[x \mapsto 3]) \quad \text{par } (3) \\
& \to & (y := 1 \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 3]) \quad \text{par } (5) \\
& \to & (c_2 \cdot \epsilon, \rho_0[x \mapsto 3, y \mapsto 1]) \quad \text{par } (3) \\
& \to & (c_3 \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 3, y \mapsto 1]) \quad \text{par } (8) \\
& \to & (y := y + x \cdot x := x + (-1) \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 3, y \mapsto 1]) \quad \text{par } (5) \\
& \to & (x := x + (-1) \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 3, y \mapsto 4]) \quad \text{par } (3) \\
& \to & (c_2 \cdot \epsilon, \rho_0[x \mapsto 2, y \mapsto 4]) \quad \text{par } (3) \\
& \to & (c_3 \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 2, y \mapsto 4]) \quad \text{par } (8) \\
& \to & (y := y + x \cdot x := x + (-1) \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 2, y \mapsto 4]) \quad \text{par } (5) \\
& \to & (x := x + (-1) \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 2, y \mapsto 6]) \quad \text{par } (3) \\
& \to & (c_2 \cdot \epsilon, \rho_0[x \mapsto 1, y \mapsto 6]) \quad \text{par } (3) \\
& \to & (c_3 \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 1, y \mapsto 6]) \quad \text{par } (8) \\
& \to & (y := y + x \cdot x := x + (-1) \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 1, y \mapsto 6]) \quad \text{par } (5) \\
& \to & (x := x + (-1) \cdot c_2 \cdot \epsilon, \rho_0[x \mapsto 1, y \mapsto 7]) \quad \text{par } (3) \\
& \to & (c_2 \cdot \epsilon, \rho_0[x \mapsto 0, y \mapsto 7]) \quad \text{par } (3) \\
& \to & (\epsilon, \rho_0[x \mapsto 0, y \mapsto 7]) \quad \text{par } (9)
\end{array}
$$

Le dernier état est un état *terminal* : aucune règle ne s'y applique. On dit que le programme c_0 *termine* en partant de $(c_0 \cdot \epsilon, \rho_0)$.

Exercice 1 (Progrès) *Montrer que les états terminaux sont exactement les états de la forme (ϵ, ρ) pour $\rho \in Env$. Ceci est une propriété de* progrès *: la sémantique de la figure 2 propose toujours une façon de continuer l'exécution tant qu'il reste quelque chose à faire (que le contexte n'est pas vide).*

Exercice 2 *Donner un exemple de commande c telle que la plus grande trace d'exécution partant de $(c \cdot \epsilon, \rho)$ est infinie.*

Exercice 3 (Déterminisme) *Montrer que la sémantique de la figure 2 est* déterministe *: pour tout état $(c \cdot C, \rho)$, il existe au plus un état $(c' \cdot C', \rho')$ tel que $(c \cdot C, \rho) \to (c' \cdot C', \rho')$. Ce ne sera pas le cas de tous les langages !*

1.4 La sémantique opérationnelle à grands pas de IMP

Une sémantique intermédiaire entre une sémantique opérationnelle à petits pas, qui décrit toutes les étapes du calcul, et une sémantique dénotationnelle, qui décrit juste le résultat final, est la sémantique opérationnelle *à grands pas*, aussi appelée *sémantique naturelle*.

On y définit des *jugements* exprimant, comme en sémantique dénotationnelle, le résultat final d'un calcul. Mais on les définit par des règles. La notion de dérivation y est ainsi fondamentale.

Dans le cas de IMP, on pourra par exemple définir nos jugements comme des triplets (ρ, c, ρ') d'un environnement ρ avant l'exécution de la commande, de la commande c elle-même, et de l'environnement obtenu après l'exécution de la commande.

$$\frac{}{\rho \vdash x := e \Rightarrow \rho[x \mapsto [\![e]\!]\,\rho]}\,(:=) \qquad \frac{}{\rho \vdash \texttt{skip} \Rightarrow \rho}\,(\texttt{skip})$$

$$\frac{\rho \vdash c_1 \Rightarrow \rho' \qquad \rho' \vdash c_2 \Rightarrow \rho''}{\rho \vdash c_1; c_2 \Rightarrow \rho''}\,(Seq)$$

$$\frac{\rho \vdash c_1 \Rightarrow \rho'}{\rho \vdash \texttt{if}\, e\, \texttt{then}\, c_1\ \texttt{else}\ c_2 \Rightarrow \rho'}\,(\texttt{if}_1) \quad \text{si } [\![e]\!]\,\rho \neq 0 \qquad \frac{\rho \vdash c_2 \Rightarrow \rho'}{\rho \vdash \texttt{if}\, e\, \texttt{then}\, c_1\ \texttt{else}\ c_2 \Rightarrow \rho'}\,(\texttt{if}_2) \quad \text{si } [\![e]\!]\,\rho = 0$$

$$\frac{\rho \vdash c \Rightarrow \rho' \qquad \rho' \vdash \texttt{while}\, e\, \texttt{do}\, c \Rightarrow \rho''}{\rho \vdash \texttt{while}\, e\, \texttt{do}\, c \Rightarrow \rho''}\,(\texttt{while}) \quad \text{si } [\![e]\!]\,\rho \neq 0 \qquad \frac{}{\rho \vdash \texttt{while}\, e\, \texttt{do}\, c \Rightarrow \rho}\,(\texttt{while}_{fin}) \quad \text{si } [\![e]\!]\,\rho = 0$$

FIGURE 3 – La sémantique opérationnelle à grands pas de IMP

La tradition est de noter les jugements à l'aide d'une syntaxe qui évoque un peu plus clairement ce qu'ils signifient, et nous utiliserons donc la notation $\rho \vdash c \Rightarrow \rho'$: « partant de ρ, l'exécution de c termine sur l'environnement ρ' ».

Un jugement tel quel n'est ni vrai ni faux : c'est juste un triplet. On définit le fait qu'un jugement soit *dérivable* dans un système de déduction. Pour IMP, on considérera celui de la figure 3.

Le style d'écriture ressemble à celui de la sémantique dénotationnelle… à la différence importante qu'il n'y a *pas* de problème avec la sémantique du `while`.

En effet, si l'on regarde la règle (`while`), nous ne sommes *pas* en train de définir la sémantique de `while` e `do` c (partant de ρ) en fonction de la sémantique de `while` e `do` c (partant de ρ', seconde prémisse). Ce que nous écrivons, c'est : si on peut fabriquer une dérivation de $\rho' \vdash$ `while` e `do` $c \Rightarrow \rho''$ (et une de $\rho \vdash c \Rightarrow \rho'$), alors on peut, via la règle (`while`), en déduire une dérivation de $\rho \vdash$ `while` e `do` $c \Rightarrow \rho''$.

La lecture de ces règles est déroutante au premier abord. Un sens adéquat de lecture, qui correspondrait en gros à l'ordre d'évaluation, serait de :

— partir de l'environnement à gauche du signe $\vdash$ (« thèse ») dans la conclusion de la règle ;
— remonter et lire les prémisses, de gauche à droite ;
— redescendre à droite du signe $\vdash$ dans la conclusion.

Par exemple, pour la règle (Seq) : partant de l'environnement ρ (première étape), on retrouve ρ à gauche de la première prémisse, qui dit d'évaluer c_1 ; une fois ceci fait, on obtient un nouvel environnement ρ', que l'on utilise pour évaluer c_2 ; ceci fournit l'environnement final ρ'', que l'on retourne à droite de la conclusion, en bas de la règle.

1.5 Rapports entre sémantiques : correction, adéquation

Relions maintenant les deux dernières sémantiques. Pour un contexte $C = c_1 \cdot c_2 \cdot c_3 \cdot \cdots \cdot c_n \cdot \epsilon$, notons $C^;$ la commande $c_1; (c_2; (c_3; \cdots; (c_n; \texttt{skip})))$. En particulier, $(c \cdot C)^; = c; C^;$.

Ceci nous donnera l'occasion de voir trois techniques de raisonnement de base sur les dérivations : l'analyse par cas, l'inversion, et le raisonnement par récurrence sur les dérivations.

Proposition 1 *Si $(C_1, \rho_1) \to (C_2, \rho_2)$ dans la sémantique à petits pas de la figure 2, alors : pour tout environnement ρ_∞ tel que $\rho_2 \vdash C_2^; \Rightarrow \rho_\infty$ est dérivable en sémantique à grands pas, $\rho_1 \vdash C_1^; \Rightarrow \rho_\infty$ est aussi dérivable en sémantique à grands pas.*

Démonstration. On fait une *analyse de cas*, et on construit la dérivation de $\rho_1 \vdash C_1^; \Rightarrow \rho_\infty$ à partir de la dérivation π donnée de $\rho_2 \vdash C_2^; \Rightarrow \rho_\infty$.

— Si la règle de sémantique à petits pas est (3), en reprenant les notations de la figure 2, c'est que C_1 est de la forme $x := e \cdot C$, $\rho_1 = \rho$, $C_2 = C$, et $\rho_2 = \rho[x \mapsto [\![e]\!]\,\rho]$. Par hypothèse, π est une dérivation de $\rho[x \mapsto [\![e]\!]\,\rho] \vdash C^; \Rightarrow \rho_\infty$. On peut donc construire la dérivation :

$$\dfrac{\dfrac{}{\rho \vdash x := e \Rightarrow \rho[x \mapsto [\![e]\!]\,\rho]}(:=) \qquad \begin{array}{c}\vdots\,\pi\\ \rho[x \mapsto [\![e]\!]\,\rho] \vdash C^; \Rightarrow \rho_\infty\end{array}}{\rho \vdash x := e; C^; \Rightarrow \rho_\infty}(Seq)$$

— Règle (4) (`skip`). On construit :

$$\dfrac{\dfrac{}{\rho \vdash \texttt{skip} \Rightarrow \rho}(\texttt{skip}) \qquad \begin{array}{c}\vdots\,\pi\\ \rho \vdash C^; \Rightarrow \rho_\infty\end{array}}{\rho \vdash \texttt{skip}; C^; \Rightarrow \rho_\infty}(Seq)$$

— Règle (8) (`while`, cas où l'on continue la boucle). C'est le cas le plus compliqué. Nous disposons d'une dérivation π de $\rho \vdash c; (\texttt{while}\, e\, \texttt{do}\, c; C^;) \Rightarrow \rho_\infty$.
Cette dérivation π est un arbre fini qui se termine nécessairement par une instance de la règle (Seq). En effet, c'est la seule règle dont la conclusion porte sur une commande qui est une séquence (le ';'). On en conclut que π est nécessairement de la forme :

$$\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \rho \vdash c \Rightarrow \rho'\end{array} \qquad \begin{array}{c}\vdots\,\pi_1'\\ \rho' \vdash \texttt{while}\, e\, \texttt{do}\, c; C^; \Rightarrow \rho_\infty\end{array}}{\rho \vdash c; (\texttt{while}\, e\, \texttt{do}\, c; C^;) \Rightarrow \rho_\infty}(Seq)$$

pour un certain environnement ρ', et certaines dérivations π_1, π_2. Cette forme de raisonnement, qui n'est autre qu'une variante du raisonnement par analyse de cas (où nous avons exclu tous les cas impossibles) s'appelle un raisonnement par *inversion*.

Par une nouvelle inversion sur la forme de la conclusion de π_1', on peut même dire que π est nécessairement de la forme :

$$\dfrac{\dfrac{\vdots\,\pi_1}{\rho \vdash c \Rightarrow \rho'} \qquad \dfrac{\vdots\,\pi_2 \qquad\qquad \vdots\,\pi_3}{\rho' \vdash \texttt{while}\, e\, \texttt{do}\, c \Rightarrow \rho'' \quad \rho'' \vdash C^; \Rightarrow \rho_\infty}}{\rho \vdash c; (\texttt{while}\, e\, \texttt{do}\, c; C^;) \Rightarrow \rho_\infty}\,(Seq)$$

On peut donc fabriquer la dérivation :

$$\dfrac{\dfrac{\vdots\,\pi_1 \qquad\qquad \vdots\,\pi_2}{\rho \vdash c \Rightarrow \rho' \quad \rho' \vdash \texttt{while}\, e\, \texttt{do}\, c \Rightarrow \rho''}\,(\texttt{while}) \qquad \vdots\,\pi_3}{\dfrac{\rho \vdash \texttt{while}\, e\, \texttt{do}\, c \Rightarrow \rho'' \qquad\qquad \rho'' \vdash C^; \Rightarrow \rho_\infty}{\rho \vdash \texttt{while}\, e\, \texttt{do}\, c; C^; \Rightarrow \rho_\infty}\,(Seq)}$$

— Nous laissons les autres cas en exercice. □

Exercice 4 *Traiter les derniers cas de la démonstration de la proposition 1.*

Notons $\rightarrow^*$ la clôture réflexive transitive de la relation $\rightarrow$. On en déduit que la sémantique à grands pas est *correcte* par rapport à la sémantique à petits pas, au sens suivant.

Théorème 2 (Correction grands pas vs. petits pas) *Si $(C, \rho) \rightarrow^* (\epsilon, \rho_\infty)$ dans la sémantique à petits pas de la figure 2, alors $\rho \vdash C^; \Rightarrow \rho_\infty$ est dérivable dans la sémantique à grands pas.*

Démonstration. Par *récurrence* sur la longueur n d'une trace d'exécution de (C, ρ) à (ϵ, ρ_∞).

Si $n = 0$, alors $\rho = \rho_\infty$ et $C^; = \texttt{skip}$. On peut donc produire la dérivation :

$$\dfrac{}{\rho_\infty \vdash \texttt{skip} \Rightarrow \rho_\infty}\,(\texttt{skip})$$

Sinon, c'est que la trace d'exécution est de la forme $(C, \rho) \rightarrow \underbrace{(C_2, \rho_2) \rightarrow^* (\epsilon, \rho_\infty)}_{\text{longueur } n-1}$. Par hypothèse de récurrence on peut dériver le jugement $\rho_2 \vdash C_2 \Rightarrow \rho_\infty$. On peut donc aussi dériver $\rho \vdash C \Rightarrow \rho_\infty$ par la proposition 1. □

On souhaite maintenant démontrer la réciproque. Ceci se fera par *récurrence sur les dérivations*. En première approche, on pourra considérer que ce principe est juste le principe de récurrence (complète) sur la taille des dérivations : pour démontrer une propriété sur les dérivations π, il suffit de la démontrer en la supposant déjà vraie sur toutes les dérivations plus petites que π.

Ce principe est valide parce que les dérivations sont des arbres *finis*.

En pratique, le principe de récurrence *structurelle* sur les dérivations suffit souvent : pour démontrer une propriété sur les dérivations π, il suffit de la démontrer en la supposant déjà vraie sur toutes les dérivations qui aboutissent aux prémisses de la dernière règle appliquée dans π.

Théorème 3 (Correction petits pas vs. grands pas) *Si $\rho \vdash c \Rightarrow \rho_\infty$ est dérivable dans la sémantique à grands pas, alors $(c \cdot \epsilon, \rho) \to^* (\epsilon, \rho_\infty)$ dans la sémantique à petits pas de la figure 2.*

Démonstration. Nous nous donnons pour cela une dérivation π de $\rho \vdash c \Rightarrow \rho_\infty$, et construisons une trace d'exécution finie de $(c \cdot \epsilon, \rho)$ à (ϵ, ρ_∞), par récurrence sur π.

Une des difficultés des raisonnements par récurrence est qu'il est parfois nécessaire de prouver un résultat général. C'est le cas ici : si nous nous en tenions à ne démontrer que $(c, \rho) \to^* (\epsilon, \rho_\infty)$, nous serions bloqués en arrivant au cas de la séquence.

Démontrons donc (toujours par récurrence sur π) le résultat plus général suivant : *pour tout contexte C*, il existe une trace d'exécution finie de $(c \cdot C, \rho)$ à (C, ρ_∞). Nous effectuons maintenant une analyse de cas sur la dernière règle utilisée :

— Cas $(:=)$. Alors c est de la forme $x := e$ et $\rho_\infty = \rho[x \mapsto [\![e]\!]\,\rho]$. On obtient la trace d'exécution $(x := e \cdot C, rho) \to (C, \rho[x \mapsto [\![e]\!]\,\rho])$, en une étape, par la règle (3).

— Cas (Seq). Ici c est de la forme $c_1; c_2$, et π est de la forme :

$$\dfrac{\begin{array}{cc}\vdots\,\pi_1 & \vdots\,\pi_2 \\ \rho \vdash c_1 \Rightarrow \rho' & \rho' \vdash c_2 \Rightarrow \rho_\infty\end{array}}{\rho \vdash c_1; c_2 \Rightarrow \rho_\infty}(Seq)$$

Par *hypothèse de récurrence* sur π_2, on obtient une trace d'exécution finie de $(c_2 \cdot C, \rho')$ à (C, ρ_∞).

Par hypothèse de récurrence sur π_1, mais cette fois-ci *avec le contexte $c_2 \cdot C$ au lieu de C* (d'où l'intérêt de quantifier universellement sur le contexte C), on obtient une trace d'exécution finie de $(c_1 \cdot c_2 \cdot C, \rho)$ à $(c_2 \cdot C, \rho')$.

En concaténant ces deux traces d'exécutions, on obtient une trace d'exécution finie de $(c_1 \cdot c_2 \cdot C, \rho)$ à (C, ρ_∞).

On complète ceci par la règle (5), qui nous donne $(c_1; c_2 \cdot C, \rho) \to (c_1 \cdot c_2 \cdot C, \rho) \to^* (C, \rho_\infty)$.

— Les autres cas sont laissés en exercice au lecteur. □

Exercice 5 *Traiter les derniers cas de la démonstration du théorème 3.*

En combinant les deux théorèmes ci-dessus, on a le résultat attendu que les deux sémantiques, à petits pas et à grands pas, calculent la même chose… du moins pour les calculs qui *terminent*.

Théorème 4 (Adéquation) *Les deux propriétés suivantes sont équivalentes :*

1. *$\rho \vdash c \Rightarrow \rho_\infty$ est dérivable dans la sémantique à grands pas ;*
2. *$(c \cdot \epsilon, \rho) \to^* (\epsilon, \rho_\infty)$ dans la sémantique à petits pas de la figure 2.*

Exercice 6 (Non-terminaison) *Que se passe-t-il pour les calculs qui ne terminent pas ? Précisément, supposez que la trace d'exécution maximale $q_0 = (c \cdot \epsilon, \rho) \to q_1 \to \cdots \to q_n \to \cdots$ partant de $(c \cdot \epsilon, \rho)$ est infinie, et montrez qu'il n'y a* aucune *dérivation d'un jugement de la forme $\rho \vdash c \Rightarrow \rho_\infty$ pour aucun environnement ρ_∞.*

Exercice 7 *On cherche maintenant à relier les deux sémantiques à petits pas. Montrer que :*

1. *si l'on peut dériver $(c, \rho) \to \rho'$ dans la sémantique à petits pas de la figure 1, alors on a $(c \cdot C, \rho) \to (C, \rho')$ pour tout contexte C dans la sémantique à petits pas de la figure 2 ;*
2. *si l'on peut dériver $(c, \rho) \to (c', \rho')$ dans la sémantique à petits pas de la figure 1, alors on a $(c \cdot C, \rho) \to^* \leftarrow^*_{Seq} (c' \cdot C, \rho')$ pour tout contexte C dans la sémantique à petits pas de la figure 2 ; la relation $\to_{Seq}$ est celle définie par l'unique règle (Seq), et ce que l'on demande de démontrer est qu'il existe un état intermédiaire q tel que $(c \cdot C, \rho) \to^* q \leftarrow^*_{Seq} (c' \cdot C, \rho')$;*
3. *si l'on a $(C_1, \rho_1) \to (C_2, \rho_2)$ dans la sémantique à petits pas de la figure 2, alors on peut dériver $(C_1^;, \rho_1) \to (C_2^;, \rho_2)$ dans la sémantique à petits pas de la figure 1 ;*
4. *si l'on a $(C_1, \rho_1) \to (\epsilon, \rho_2)$ dans la sémantique à petits pas de la figure 2, alors on peut dériver $(C_1^;, \rho_1) \to \rho_2$ dans la sémantique à petits pas de la figure 1.*

1.6 Une sémantique de C−−

Nous allons donner une sémantique à grands pas de C−−. Même si la lecture de certaines des règles a pu vous dérouter, il se trouve que ce sera le formalisme permettant la description la plus synthétique de la sémantique.

Le language C−− travaille sur des valeurs non plus dans $\mathbb{Z}$, mais dans $\mathbb{Z}_k$, avec k un multiple de 8 strictement positif : $k = 32$ ou $k = 64$, typiquement. On définit $\mathbb{Z}_k$ comme étant l'intervalle des entiers compris entre -2^{k-1} et $2^{k-1} - 1$: on reconnaît le domaine des entiers *signés* sur k bits.

Le fait que C−− utilise aussi des pointeurs nécessite d'introduire une notion de *mémoire* μ, c'est-à-dire une fonction (partielle) qui à chaque adresse $a \in Addr$ (où $Addr$, l'ensemble des adresses, est l'ensemble des éléments des entiers relatifs multiples de $k/8$) pour laquelle $\mu(a)$ est définie associe une valeur $\mu(a) \in \mathbb{Z}_k$. On notera $\mathrm{dom}\ \mu$ l'ensemble des adresses a telles que $\mu(a)$ est définie.

Un environnement ρ associe désormais à chaque variable x une *adresse* $a = \rho(x)$ où est stockée la valeur de x. En particulier, elle ne stocke plus la valeur de x directement. Pour l'obtenir, il faudra calculer $\mu(\rho(x))$.

Nous changeons un peu la notion d'environnement : désormais, un environnement sera aussi une fonction *partielle*. Les variables hors du domaine de l'environnement ρ seront celles qui n'ont pas été déclarées.

Un autre changement par rapport à IMP est que les commandes *et les* expressions peuvent avoir des effets de bord. Nous devrons donc inclure à la fois ρ et μ dans l'état.

Finalement, on décrira les expressions et commandes à évaluer dans la syntaxe des arbres de syntaxe abstraite Caml qui les décrivent. On omettra les « `location` » (qui servent d'aide aux messages d'erreur), et on identifiera donc les « `loc_expr` » et les « `expr` », les `loc_code` » et les « `code` ».

Un *programme* π est alors juste un objet du type `var_declaration list` du projet : une liste de :

— déclarations de variables `CDECL` x (pour un nom de variable x ; équivalent à la déclaration C `int` x) ;
— définitions de fonctions `CFUN`(f, ℓ, c) : ceci est équivalent à la déclaration C :
`int` f `(`ℓ`) {`
c
`}`
La liste ℓ est elle aussi de type `var_declaration list`, mais est *garantie* ne contenir que des déclarations de variables `CDECL` x_1, ..., `CDECL` x_n, et aucune définition de fonction (ce qui n'aurait aucun sens, au passage).
On aura la même garantie dans les blocs `CBLOCK`$(\ell, [c_1; c_2; \cdots; c_n])$, où ℓ ne contiendra que des déclarations de variables. Ces derniers sont les équivalents du C : `{` ℓ c_1 ; c_2 ; ... ; c_n : `}`, où ℓ est une déclaration de variables locales (éventuellement vide, le bloc servant alors à coder la séquence).

Dans la suite,

on supposera un programme π fixé une fois pour toutes.

Ceci nous évitera de devoir écrire des jugements de la forme $\rho, \mu \vdash^{\pi} c \Rightarrow \rho', \mu'$: on s'économisera l'exposant π, et donc un peu de la lourdeur de la notation. C'est d'autant plus indiqué que cet indice ne serait utile que dans une règle : la règle (`CALL`) que nous verrons plus bas, et qui gère l'appel des fonctions.

On supposera de plus que π nous permet de définir un environnement ρ_{glob}, l'*environnement des variables globales*. Son domaine est l'ensemble des variables x telles que `CDECL` x est dans la liste π. On suppose que ρ_{glob} est *injective*, c'est-à-dire attribue à deux variables globales distinctes des adresses distinctes.

Cette sémantique ne gère pas la compilation séparée. Dans un souci de simplicité, on considérera que le programme est la concaténation de *tous* les fichiers sources le composant... y compris les sources des fonctions de la bibliothèque C standard comme `fprintf` ou `exit`, y compris les déclarations de variables globales comme `stderr`.

Expressions et l-values. On commence par définir des règles exprimant comment calculer l'adresse des expressions qui ont une adresse (les *l-values*, le « l » abrégeant « left », la gauche du symbole d'affectation).

La forme du jugement associé sera $\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'$, où a est l'adresse souhaitée, et μ' la mémoire après l'exécution de l'évaluation de l'adresse de e. On définit donc :

$$\frac{}{\rho, \mu \vdash_{\&} \texttt{VAR}\ x \Rightarrow \rho(x), \mu} \ (\&\texttt{VAR}) \quad \text{si } x \in \text{dom } \rho$$

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow a, \mu''}{\rho, \mu \vdash_{\&} \texttt{OP2}(\texttt{S\_INDEX}, e_1, e_2) \Rightarrow a + n.(k/8), \mu''} \ (\&\texttt{INDEX}) \quad \text{si } n \in \mathbb{Z}_k, a \in Addr$$

Les deux prémisses de cette dernière règle affichent '$\vdash$' et non '$\vdash_{\&}$'. Ce n'est pas une erreur.

Notez aussi que l'on évalue *de droite à gauche* : c'est un choix de conception de C$--$ (que ne fait pas C), et qui est là pour vous simplifier la vie.

Les affectations. On peut affecter des variables ou des éléments de tableaux uniquement, en C$--$.

$$\frac{\rho, \mu \vdash e \Rightarrow V, \mu' \qquad \rho, \mu' \vdash_{\&} x \Rightarrow a, \mu''}{\rho, \mu \vdash \texttt{SET\_VAR}(x, e) \Rightarrow V, \mu''[a \mapsto V]} \ (\texttt{SET\_VAR}) \quad \text{si } a \in \text{dom } \mu''$$

La condition de bord $a \in \text{dom } \mu''$ sert à assurer qu'une affectation se fasse dans une zone de mémoire préalablement allouée.

$$\frac{\rho, \mu \vdash e_2 \Rightarrow V, \mu' \qquad \rho, \mu' \vdash_{\&} \texttt{OP2}(\texttt{S\_INDEX}, \texttt{VAR}\ x, e_1) \Rightarrow a, \mu''}{\rho, \mu \vdash \texttt{SET\_ARRAY}(x, e_1, e_2) \Rightarrow V, \mu''[a \mapsto V]} \ (\texttt{SET\_ARRAY}) \quad \text{si } a \in \text{dom } \mu''$$

Variables, constantes. La règle (Rd) permet de lire la valeur d'une l-value. Ceci traite du cas des variables et des accès aux tableaux.

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash e \Rightarrow \mu'(a), \mu'} \ (Rd) \quad \text{si } a \in \text{dom } \mu'$$

Pour $n \in \mathbb{Z}$, on notera $n \bmod 2^k$ l'unique élément de $\mathbb{Z}_k$ congru à n modulo 2^k.

$$\frac{}{\rho, \mu \vdash \texttt{CST}\ n \Rightarrow n \bmod 2^k, \mu} \ (\texttt{CST})$$

On a défini la mémoire μ comme un tableau de mots de 2^k bits. Nous pouvons aussi la voir comme un tableau d'octets, comme suit. Notons μ_{oct} la fonction partielle de Z_k vers l'ensemble des octets de domaine $\text{dom } \mu_{oct} = \{a \in Addr \mid (k/8)\lfloor a/(k/8)\rfloor \in \text{dom } \mu\}$, et telle que pour tout $a \in \text{dom } \mu$, $\mu(a) = \sum_{i=0}^{k/8-1} \mu_{oct}(a+i)256^i$.

Exercice 8 *Cette vue de la mémoire est-elle petit-boutiste ? grand-boutiste ?*

Soit s une chaîne de caractères, disons formée des caractères s_0, s_1, …, s_{n-1} dans cet ordre. Nous dirons que la mémoire μ *contient* s *à l'adresse* a si et seulement si les adresses a, $a+1$, …, $a+n-1$, $a+n$ sont toutes dans $\text{dom } \mu_{oct}$, $\mu_{oct}(a) = s_0$, $\mu_{oct}(a+1) = s_1$, …, $\mu_{oct}(a+n-1) = s_{a+n-1}$, et $\mu_{oct}(a+n) = 0$.

$$\frac{}{\rho, \mu \vdash \texttt{STRING}\ s \Rightarrow a, \mu} \ (\texttt{STRING}) \quad \text{si } \mu \text{ contient } s \text{ à l'adresse } a$$

Cette règle peut paraître un peu curieuse : comment assure-t-on qu'il y a une adresse où la chaîne s serait stockée ? S'il y en a plusieurs, laquelle retourne-t-on ?

La réponse est qu'on laisse toute liberté au compilateur pour trouver une telle adresse. S'il n'y en a pas, la sémantique du programme sera indéfinie. En pratique, on s'arrangera pour préallouer la chaîne de caractères s à une adresse a, et le compilateur produire du code assembleur qui se contente de fournir cette adresse.

Il est hors de question d'*allouer* l'adresse a au moment de l'évaluation de `STRING` s : ce n'est pas ce que nous souhaitons que ce genre d'expression fasse. On veut juste retourner l'adresse d'une zone mémoire où la chaîne s est *déjà* stockée.

Les opérations monadiques. Pour alléger un peu l'écriture des règles, nous noterons n, n_1, n_2, …, des éléments de $\mathbb{Z}_k$, a, a_1, a_2, …, des éléments de $Addr$, et V, V_1, V_2, …, des valeurs pouvant être des éléments de $\mathbb{Z}_k$ ou des adresses.

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_MINUS}, e) \Rightarrow (-n) \bmod 2^k, \mu'} \ (\texttt{MINUS})$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_NOT}, e) \Rightarrow (-1-n) \bmod 2^k, \mu'} \ (\texttt{NOT})$$

Exercice 9 *À quoi peut-on voir que les entiers sont codés en complément à deux dans cette sémantique ?*

Voici la sémantique de la post-incrémentation ; en syntaxe concrète, `x++` ou `a[i]++`, par exemple.

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_POST\_INC}, e) \Rightarrow n, \mu'[a \mapsto (n+1) \bmod 2^k]} \ (\texttt{POST\_INC}) \quad \text{si } a \in \operatorname{dom} \mu', n = \mu'(a)$$

De même, la post-décrémentation, `x--` ou `a[i]--` :

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_POST\_DEC}, e) \Rightarrow n, \mu'[a \mapsto (n-1) \bmod 2^k]} \ (\texttt{POST\_DEC}) \quad \text{si } a \in \operatorname{dom} \mu', n = \mu'(a)$$

De façon symétrique, la pré-incrémentation, `++x` ou `++a[i]` :

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_PRE\_INC}, e) \Rightarrow (n+1) \bmod 2^k, \mu'[a \mapsto (n+1) \bmod 2^k]} \ (\texttt{PRE\_INC}) \quad \text{si } a \in \operatorname{dom} \mu', n = \mu'(a)$$

et la pré-décrémentation, `--x` ou `--a[i]` :

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_PRE\_DEC}, e) \Rightarrow (n-1) \bmod 2^k, \mu'[a \mapsto (n-1) \bmod 2^k]} \ (\texttt{PRE\_DEC}) \quad \text{si } a \in \operatorname{dom} \mu', n = \mu'(a)$$

Les (autres) opérations binaires. Nous avons déjà traité de l'opération `S_INDEX`, au travers des règles (`&INDEX`) et (Rd). Regardons les autres.

On évalue toujours de droite à gauche en C−−.

$$\frac{\rho,\mu \vdash e_2 \Rightarrow n_2,\mu' \qquad \rho,\mu' \vdash e_1 \Rightarrow n_1,\mu''}{\rho,\mu \vdash \texttt{OP2}(\texttt{S\_ADD},e_1,e_2) \Rightarrow (n_1+n_2) \bmod 2^k,\mu''}\ (\texttt{ADD})$$

$$\frac{\rho,\mu \vdash e_2 \Rightarrow n_2,\mu' \qquad \rho,\mu' \vdash e_1 \Rightarrow n_1,\mu''}{\rho,\mu \vdash \texttt{OP2}(\texttt{S\_SUB},e_1,e_2) \Rightarrow (n_1-n_2) \bmod 2^k,\mu''}\ (\texttt{SUB})$$

$$\frac{\rho,\mu \vdash e_2 \Rightarrow n_2,\mu' \qquad \rho,\mu' \vdash e_1 \Rightarrow n_1,\mu''}{\rho,\mu \vdash \texttt{OP2}(\texttt{S\_MUL},e_1,e_2) \Rightarrow (n_1\times n_2) \bmod 2^k,\mu''}\ (\texttt{MUL})$$

Pour la division, définissons $\lfloor x \rceil$ pour chaque nombre réel x comme étant l'entier de même signe que x dont la valeur absolue est la partie entière de la valeur absolue de x. Donc $\lfloor 2{,}3 \rceil = 2$, mais $\lfloor -2{,}3 \rceil = -2$.

$$\frac{\rho,\mu \vdash e_2 \Rightarrow n_2,\mu' \qquad \rho,\mu' \vdash e_1 \Rightarrow n_1,\mu''}{\rho,\mu \vdash \texttt{OP2}(\texttt{S\_DIV},e_1,e_2) \Rightarrow \lfloor n_1/n_2 \rceil \bmod 2^k,\mu''}\ (\texttt{DIV})\quad \text{si } n_2 \neq 0$$

$$\frac{\rho,\mu \vdash e_2 \Rightarrow n_2,\mu' \qquad \rho,\mu' \vdash e_1 \Rightarrow n_1,\mu''}{\rho,\mu \vdash \texttt{OP2}(\texttt{S\_MOD},e_1,e_2) \Rightarrow (n_1 - n_2\times\lfloor n_1/n_2 \rceil) \bmod 2^k,\mu''}\ (\texttt{MOD})\quad \text{si } n_2 \neq 0$$

Exercice 10 *Soit $n_2 \neq 0$. Montrer que $n_1 - n_2 \times \lfloor n_1/n_2 \rceil$ est le reste de la division de $|n_1|$ par $|n_2|$ si $n_1 \geq 0$, et est l'opposé de ce reste si $n_1 \leq 0$.*

Les bizarreries dans la sémantique de la division et du reste sont, bien sûr, faites pour correspondre à la sémantique des mêmes opérations en assembleur x86 (et de la plupart des autres processeurs).

Les comparaisons.

$$\frac{\rho,\mu \vdash e_2 \Rightarrow n_2,\mu' \qquad \rho,\mu' \vdash e_1 \Rightarrow n_1,\mu''}{\rho,\mu \vdash \texttt{CMP}(\texttt{C\_LT},e_1,e_2) \Rightarrow 1,\mu''}\ (\texttt{LT}_1)\quad \text{si } n_1 < n_2$$

$$\frac{\rho,\mu \vdash e_2 \Rightarrow n_2,\mu' \qquad \rho,\mu' \vdash e_1 \Rightarrow n_1,\mu''}{\rho,\mu \vdash \texttt{CMP}(\texttt{C\_LT},e_1,e_2) \Rightarrow 0,\mu''}\ (\texttt{LT}_0)\quad \text{si } n_1 \geq n_2$$

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n_2, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow n_1, \mu''}{\rho, \mu \vdash \texttt{CMP}(\texttt{C\_LE}, e_1, e_2) \Rightarrow 1, \mu''} \; (\texttt{LE}_1)$$
si $n_1 \leq n_2$

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n_2, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow n_1, \mu''}{\rho, \mu \vdash \texttt{CMP}(\texttt{C\_LE}, e_1, e_2) \Rightarrow 0, \mu''} \; (\texttt{LE}_0)$$
si $n_1 > n_2$

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n_2, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow n_1, \mu''}{\rho, \mu \vdash \texttt{CMP}(\texttt{C\_EQ}, e_1, e_2) \Rightarrow 1, \mu''} \; (\texttt{EQ}_1)$$
si $n_1 = n_2$

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n_2, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow n_1, \mu''}{\rho, \mu \vdash \texttt{CMP}(\texttt{C\_EQ}, e_1, e_2) \Rightarrow 0, \mu''} \; (\texttt{EQ}_0)$$
si $n_1 \neq n_2$

$$\frac{\rho, \mu \vdash e_1 \Rightarrow n, \mu' \qquad \rho, \mu' \vdash e_2 \Rightarrow V, \mu''}{\rho, \mu \vdash \texttt{EIF}(e_1, e_2, e_3) \Rightarrow V, \mu''} \; (\texttt{EIF}_1)$$
si $n \neq 0$

$$\frac{\rho, \mu \vdash e_1 \Rightarrow 0, \mu' \qquad \rho, \mu' \vdash e_3 \Rightarrow V, \mu''}{\rho, \mu \vdash \texttt{EIF}(e_1, e_2, e_3) \Rightarrow V, \mu''} \; (\texttt{EIF}_0)$$

Les expressions séquences. Les expressions séquences sont des expressions de la forme $e_1, e_2, \cdots, e_n$, où les e_i sont elles-mêmes des expressions. Elles s'évaluent de gauche à droite, et retournent la valeur de e_n si $n \neq 0$, et une valeur arbitraire si $n = 0$.

$$\frac{}{\rho, \mu \vdash \texttt{ESEQ}\ [] \Rightarrow V, \mu} \; (ESeq0)$$

$$\frac{\rho, \mu \vdash e \Rightarrow V, \mu' \qquad \rho, \mu' \vdash \texttt{ESEQ}\ L \Rightarrow V', \mu''}{\rho, \mu \vdash \texttt{ESEQ}\ (e :: L) \Rightarrow V', \mu''} \; (ESeq1)$$

Les appels de fonctions. Il y a beaucoup à expliquer ici.

D'abord, encore une fois, on évalue les arguments de droite à gauche.

Ensuite, on a besoin d'allouer de la mémoire, pour allouer la place (sur la pile dans votre compilateur) pour les paramètres de la fonction f. Pour ceci, on suppose une fonction $alloc$ qui a chaque mémoire μ associe, soit une adresse $a \in Addr$ qui n'est pas dans $\mathrm{dom}\ \mu$ (qui n'est pas

encore allouée), soit le symbole spécial $\bot$, supposé hors de $Addr$ (ceci représente typiquement l'épuisement de l'espace mémoire disponible).

$$\frac{\begin{array}{rcl}\rho, \mu & \vdash & e_k \Rightarrow n_k, \mu_k \\ \rho, \mu_k[a_k \mapsto n_k] & \vdash & e_{k-1} \Rightarrow n_{k-1}, \mu_{k-1} \\ & \vdots & \\ \rho, \mu_2[a_2 \mapsto n_2] & \vdash & e_1 \Rightarrow n_1, \mu_1 \\ \rho_{\text{glob}}[x_1 \mapsto a_1, \cdots, x_k \mapsto a_k], \mu_1[a_1 \mapsto n_1] & \vdash & c \Rightarrow \texttt{ret}\ n, \mu'\end{array}}{\begin{array}{r}\rho, \mu \vdash \texttt{CALL}(f, [e_1; \cdots; e_k]) \Rightarrow n, \mu' \\ \text{s'il existe une unique entrée } \texttt{CFUN}(g, [x_1; \cdots; x_{k'}], c) \\ \text{avec } g = f \text{ dans } \pi \\ \text{et si } k' = k, \\ \text{et si les } x_i \text{ sont disjoints } (1 \leq i \leq k), \\ a_k = alloc(\mu_k) \neq \bot, \\ \cdots, \\ a_2 = alloc(\mu_2) \neq \bot, \\ a_1 = alloc(\mu_1) \neq \bot\end{array}}\ (\texttt{CALL})$$

On notera une nouvelle forme de jugement (la dernière prémisse), où ce que l'on évalue n'est plus une expression mais une commande (c), et l'on retourne non pas une valeur et une mémoire, mais :

— un *paquet de retour* `ret` n ; ceci sera la « valeur » de l'instruction `return e` lorsque la valeur de `e` est l'entier n ;
— et une mémoire μ'.

Commandes Commençons par l'instruction `return` avec argument (par exemple, `return x+1`) :

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{CRETURN}\ (\texttt{Some}\ e) \Rightarrow \texttt{ret}\ n, \mu'}\ (\texttt{RETURN})$$

Ensuite, l'instruction `return` sans argument :

$$\frac{}{\rho, \mu \vdash \texttt{CRETURN}\ (\texttt{None}) \Rightarrow \texttt{ret}\ n, \mu'}\ (\texttt{RETURNO})$$
$$\text{où } n \in \mathbb{Z}_k$$

qui donc retourne un entier machine *arbitraire*. (La sémantique est donc non déterministe, en particulier.) En C, la question ne se pose pas : le système de typage de C garantit qu'on ne peut écrire `return` sans argument que si la fonction courante a le type de retour `void`, ce qui garantit que quelque valeur que nous retournions, celle-ci ne sera de toute façon jamais utilisée. En C−−, il n'y a pas de typage… et nous décidons de retourner une valeur quelconque. C'est (de nouveau) fait pour vous faciliter la vie : si, comme recommandé dans le projet, vous décidez de placer la valeur retournée dans `%eax` (resp., `%rax`), ceci vous permet de ne pas vous préoccuper du contenu de ce registre du tout lors des `return` sans argument.

Pour les autres commandes, le jugement typique aura la forme $\rho, \mu \vdash c \Rightarrow \mu'$, où μ' est la mémoire obtenue lorsque l'exécution de c se termine normalement (sans passer par un `return`). Par exemple, pour les expressions vues comme commandes :

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{CEXPR}\ e \Rightarrow \mu'}\ (\texttt{CEXPR})$$

On se retrouve donc avec deux sortes de jugements pour une commande c : $\rho, \mu \vdash c \Rightarrow \mu'$ si c se termine normalement, et $\rho, \mu \vdash c \Rightarrow \texttt{ret}\ n, \mu'$ si c effectue un `return` d'une valeur n.

Pour une séquence $c_1; c_2$, nous devrons écrire que : soit c_1 effectue un `return` et l'on n'évalue pas c_2, soit c_1 se termine normalement, et on évalue c_2. Nous allons factoriser la sémantique des séquences dans une série de règles établissant des jugements de la forme $\rho, \mu \vdash L \Rightarrow \mu'$ ou $\rho, \mu \vdash L \Rightarrow \texttt{ret}\ n, \mu'$, où L est une *liste* finie de commandes, à exécuter en séquence. On note la liste vide [] comme en Caml, et celle obtenue en ajoutant c en tête de L sera notée $c :: L$. On notera res pour un *résultat*, c'est-à-dire soit une mémoire μ' (si la liste ou la commande se termine normalement), soit un couple $\texttt{ret}\ n, \mu'$ formé d'un paquet de retour et d'une mémoire (si la liste ou la commande effectue un `return`).

$$\frac{}{\rho, \mu \vdash [] \Rightarrow \mu}\ (Seq0)$$

$$\frac{\rho, \mu \vdash c \Rightarrow \mu' \qquad \rho, \mu' \vdash L \Rightarrow res}{\rho, \mu \vdash c :: L \Rightarrow res}\ (Seq1)$$

$$\frac{\rho, \mu \vdash c \Rightarrow \texttt{ret}\ n, \mu'}{\rho, \mu \vdash c :: L \Rightarrow \texttt{ret}\ n, \mu'}\ (SeqRet)$$

On peut maintenant donner la sémantique des blocs, qui allouent de la place pour des variables locales, puis effectuent une séquence de commandes.

$$\frac{\rho[x_1 \mapsto a_1, \cdots, x_k \mapsto a_k], \mu_k \vdash L \Rightarrow res}{\rho, \mu \vdash \texttt{CBLOCK}([x_1; \cdots; x_k], L) \Rightarrow res}\ (\texttt{CBLOCK})$$

$$\begin{aligned} \text{où } a_1 = alloc(\mu) \neq \bot, \\ n_1 \in \mathbb{Z}_k, \mu_1 = \mu[a_1 \mapsto n_1], \\ a_2 = alloc(\mu_1) \neq \bot, \\ n_2 \in \mathbb{Z}, \mu_2 = \mu_1[a_2 \mapsto n_2], \\ \ldots, a_k = alloc(\mu_{k-1}) \neq \bot, \\ n_k \in \mathbb{Z}, \mu_k = \mu_{k-1}[a_k \mapsto n_k] \end{aligned}$$

Ci-dessus, les entiers n_1, …, n_k ne sont soumis à aucune contrainte. Ils sont donc *quelconques* : comme en C, les variables nouvellement déclarées ne sont pas initialisées. (Il en est de même des variables globales, si vous regardez bien.)

Il ne reste plus qu'à traiter de la conditionnelle, très proche de l'autre conditionnelle `EIF` :

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu' \qquad \rho, \mu' \vdash c_1 \Rightarrow res}{\rho, \mu \vdash \texttt{CIF}(e, c_1, c_2) \Rightarrow res}\ (\texttt{CIF}_1)$$

$$\text{si } n \neq 0$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu' \qquad \rho, \mu' \vdash c_2 \Rightarrow res}{\rho, \mu \vdash \texttt{CIF}(e, c_1, c_2) \Rightarrow res} \, (\texttt{CIF}_2) \quad \text{si } n = 0$$

et des boucles `while` :

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu' \quad \rho, \mu' \vdash [c; \texttt{CWHILE}(e, c)] \Rightarrow res}{\rho, \mu \vdash \texttt{CWHILE}(e, c) \Rightarrow res} \, (\texttt{CWHILE}) \quad \text{si } n \neq 0$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{CWHILE}(e, c) \Rightarrow \mu'} \, (\texttt{CWHILE}_{fin}) \quad \text{si } n = 0$$