Notes de Révision – Leçon 930

Sémantique des Langages de Programmation. Exemples.

David Baelde

25 février 2019

Rapport du jury 2018 pour la leçon : *L'objectif est de formaliser ce qu'est un programme : introduction des sémantiques opérationnelle et dénotationnelle, dans le but de pouvoir faire des preuves de programmes, des preuves d'équivalence, des preuves de correction de traduction. Ces notions sont typiquement introduites sur un langage de programmation (impératif) jouet. On peut tout à fait se limiter à un langage qui ne nécessite pas l'introduction des CPOs et des théorèmes de point fixe généraux. En revanche, on s'attend ici à ce que les liens entre sémantique opérationnelle et dénotationnelle soient étudiés (toujours dans le cas d'un langage jouet). Il est aussi important que la leçon présente des exemples d'utilisation des notions introduites, comme des preuves d'équivalence de programmes ou des preuves de correction de programmes.*

Table des matières

1 Généralités

Un programme *est* un arbre de syntaxe abstraite (AST) ; les problèmes de parsing liés au passage d'un mot à un AST sont du ressort d'autres leçons. Le vrai sujet de la leçon est de formaliser ce que *fait* ou *signifie* un programme, c'est à dire de définir sa sémantique.

La sémantique *opérationnelle* définit comment les programmes calculent. On distingue deux styles :

— La sémantique opérationnelle *à grands pas* (ou sémantique *naturelle*) définit directement une relation entre un programme et le résultat de son calcul. On veut par exemple exprimer que $1 + (2 + 3)$ s'évalue en 6.

— La sémantique opérationnelle *à petits pas* décrit plus finement le calcul comme une succession de transformations du programme. On veut par exemple exprimer que $1 + (2 + 3)$ se réduit en $1 + 5$ puis en 6.

— On peut trouver d'autres sémantiques opérationnelles, explicitant par exemple une machine (plus ou moins abstraite) qui réalise le calcul.

La sémantique *dénotationelle* associe à tout programme une interprétation ou dénotation qui représente de façon abstraite le calcul réalisé par le programme.

Ces définitions ne sont pas formelles, et il faudra prendre soin de les interpréter de façon intéressante selon le contexte. On devra par exemple associer au programme une représentation de l'état de la mémoire. La notion de calcul va aussi varier : il peut être purement fonctionnel ou manipuler la mémoire, être déterministe ou pas, impliquer des interactions entre sous-programmes concurrents, etc. Vu le cadre de la leçon, il vaut mieux éviter les digressions liées au λ-calcul et à l'ordre supérieur, et se concentrer sur la programmation impérative. On peut même, plutôt que de définir de façon générale les types de sémantiques, simplement utiliser cette terminologie sur les exemples concrets présentés dans le plan.

Quel que soit le style, la définition d'une sémantique est guidée par la syntaxe des programmes. En sémantique opérationnelle, c'est la structure du programme qui dicte quelles réductions sont possibles (à petits pas) ou comment on peut obtenir une évaluation (à grands pas). En sémantique dénotationnelle, les programmes s'interprètent de façon *compositionnelle* : chaque construction du langage correspond à une construction sur les dénotations, e.g. $[\![p + q]\!] = [\![p]\!] \overline{+} [\![q]\!]$. La différence est qu'en sémantique opérationnelle on ne va manipuler que des expressions syntaxiques, tandis que les dénotations intéressantes vont avoir tendance à être des objets plus abstraits.

Les différentes sémantiques ont des utilisations différentes. Les sémantiques opérationnelles sont les plus simples pour définir un langage. On peut les "suivre" pour vérifier mécaniquement le résultat d'un calcul, implémenter un interprète pour le langage. Elles peuvent aussi servir à justifier que certaines transformations de programme (e.g. les optimisations d'un compilateur) préservent le résultat du calcul. Tout ceci est vrai qu'on soit à grands pas ou petits pas, mais la différence peut être importante : typiquement, seule la sémantique à petits pas

peut rendre compte de la non-terminaison du calcul ; par contre, sa finesse peut rendre les raisonnements inutilement lourds. Les sémantiques opérationnelles sont par contre généralement de mauvais outils quand il s'agit de montrer que deux programmes sont équivalents : pour cela, la sémantique dénotationnelle permet précisément de s'abstraire des détails syntaxiques pour raisonner plus efficacement.

2 Le langage Imp

On utilise dans ce document le langage Imp de Winskel [1993] qui fournit un excellent support pour la leçon. On rappelle (partiellement) les définitions de [Winskel, 1993, chap. 2] ci-dessous, en adaptant un peu la syntaxe pour faire la distinction entre les entiers naturels et leurs opérations usuelles, et leurs représentations syntaxiques.

Definition 1 (Syntaxe)**.** *On suppose un ensemble infini d'*adresses *notées X, Y, etc. On définit les expressions arithmétiques et booléennes, et les commandes, par la syntaxe abstraite suivante, où n dénote un entier relatif :*

$$\begin{array}{lcl} a & ::= & \overline{n} \mid \overline{X} \mid a \mathbin{\overline{+}} a' \mid \dots \\ b & ::= & \overline{\top} \mid \overline{\bot} \mid a \mathbin{\overline{=}} a' \mid \dots \\ c & ::= & X := a \mid \dots \end{array}$$

On notera $\equiv$ l'égalité (syntaxique) des expressions et commandes. J'ai choisi de noter les booléens $\top$ et $\bot$, ce qui n'est peut être pas idéal quand on commence à travailler sur les cpo et autres treillis complets.

La sémantique informelle de ces constructions étant claire, on peut remarquer que cette syntaxe abstraite est implicitement typée : une expression arithmétique renvoie toujours un entier, une expression booléenne un booléen, et une commande ne renvoie rien et ne fait que modifier l'état.

Definition 2 (États)**.** *Un état est une fonction totale des adresses dans $\mathbb{Z}$. Les états sont notés σ et l'ensemble des états est noté Σ. On note $\sigma[n/X]$ l'état qui associe n à X et $\sigma(Y)$ pour tout $Y \neq X$. L'état $\mathbf{0}$ est celui qui associe 0 à toute adresse.*

Un état est nécessaire pour donner un sens à une expression : c'est lui qui détermine le contenu d'une adresse. Il modélise en fait l'état de la mémoire d'un ordinateur. De ce point de vue, une fonction partielle pourrait être plus réaliste, permettant de rendre compte des *null-pointer exceptions* et autres erreurs de segmentation qui se produisent quand un programme tente d'accéder à une adresse invalide ou interdite. On préfère ici éviter de modéliser ces erreurs. C'est aussi pour cela qu'on ne considère pas la division dans les expressions arithmétiques.

3 Sémantique opérationnelle à grands pas

[Winskel, 1993, section 2.2–2.4] définit trois relations, $\langle a, \sigma\rangle \to n$, $\langle b, \sigma\rangle \to t$ et $\langle c, \sigma\rangle \to \sigma'$ où $\sigma, \sigma' \in \Sigma$, et a, b et c sont respectivement des expressions arithmétiques et booléennes, $n \in \mathbb{Z}$ et $b \in \mathbb{B}$. Les deux premières relations sont des relations d'*évaluation*, la dernière sera plutôt appelée *exécution* dans la mesure où le calcul d'une commande ne retourne pas de valeur. L'état est seulement lu dans les deux premières relations, mais aussi modifié dans la dernière.

Chaque relation est définie inductivement, comme plus petite relation close par un système de règles. Contrairement à ce qu'on fait typiquement quand on définit un système de preuve, on ne distingue pas ici le jugement syntaxique (e.g. $\langle a, \sigma\rangle \to n$) de l'énoncé qui dit que le jugement est vrai. On dira donc "si $\langle a, \sigma\rangle \to n$, alors ..." et non pas "si $\langle a, \sigma\rangle \to n$ est dérivable, alors ...". Cela ne nous empêche pas de raisonner par induction, mais l'induction portera directement sur le prédicat inductif et non sur l'arbre de dérivation d'un jugement.

On peut facilement montrer (par induction sur les expressions ou commandes) que ces relations sont en fait des fonctions, totales dans le cas des expressions.

Proposition 1. *L'évaluation des expressions arithmétiques est déterministe et totale :*

- *Pour tout a, σ, n et n' tel que $\langle a, \sigma\rangle \to n$ et $\langle a, \sigma\rangle \to n'$, on a $n = n'$.*
- *Pour tout a et σ il existe n tel que $\langle a, \sigma\rangle \to n$.*

On a un résultat analogue pour les expressions booléennes. Pour les commandes on peut montrer le caractère déterministe de l'exécution. On n'a pas la totalité, à cause de la non-terminaison.

Exemple 1. *Posons $c \equiv$ while $\overline{\top}$ do skip. Quel que soit σ il n'existe pas de σ' tel que $\langle c, \sigma\rangle \to \sigma'$. Cela se montre par induction sur $\langle c, \sigma\rangle \to \sigma'$.*

Exemple 2. *On pose $c \equiv R := 1;$ while $X \leq 1$ do $R := R \times X; X := X - 1$. Cette commande calcule la fonction factorielle dans le sens suivant :*

- *pour tout $n \in \mathbb{N}$, σ et σ' tel que $\langle c, \sigma[n/X]\rangle \to \sigma'$ on a $\sigma'(R) = n!$;*
- *pour tout $n \in \mathbb{N}$ et σ il existe σ' tel que $\langle c, \sigma[n/X]\rangle \to \sigma'[n!/R]$.*

On a séparé correction partielle et terminaison, même si les deux peuvent être montrés du même coup. Précisément, si w est la boucle while *de c on peut montrer par induction sur n que pour tout σ, $\langle w, \sigma[n/X]\rangle \to \sigma[\sigma(R) \times n!/R, 0/X]$.*

Definition 3 (Équivalence observationnelle)**.** *On écrit $c \approx c'$ quand, pour tout σ et σ', $\langle c, \sigma\rangle \to \sigma'$ ssi $\langle c', \sigma\rangle \to \sigma'$.*

La relation mérite son nom dans la mesure où, dans le langage Imp, la seule observation que l'on peut faire sur le calcul d'une commande est la façon dont elle transforme un état de départ en un état d'arrivée. Ces observations sont en effet faites par composition séquentielle. On pourrait montrer que $c \approx c'$ est équivalent au fait que pour tout contexte, $\langle C[c], \mathbf{0}\rangle$ termine [1] ssi $\langle C[c'], \mathbf{0}\rangle$

1. Cela signifie qu'il existe un σ' tel que $\langle C[c], \mathbf{0}\rangle \to \sigma'$.

termine. (Cela ne changerait rien de demander que les états d'arrivée soient les mêmes, ou qu'une certaine variable contienne 0 dans l'état d'arrivée.) La démonstration serait assez pénible en ne travaillant qu'avec la sémantique opérationnelle, mais elle deviendra évidente avec la sémantique dénotationnelle. On peut par contre facilement montrer un sens de l'équivalence :

Proposition 2. *Si $c \not\approx c'$ alors il existe un contexte $C \equiv c_0; \bullet; c_1$ tel que ou bien $\langle C[c], \mathbf{0}\rangle$ termine mais pas $\langle C[c'], \mathbf{0}\rangle$, ou bien le contraire.*

Idée de preuve. La commande c_0 construit (la partie utile de) l'état σ qui fait échouer $c \approx c'$. Si seulement l'une des deux commandes termine sur σ, on a fini. Sinon elles terminent mais sur des états $\sigma'_1 \neq \sigma'_2$, et c_1 est la commande qui termine seulement sur l'un des deux σ'_i. □

Exemple 3. *On n'a pas* $\mathsf{skip} \approx X := X + 1$. *On a* $(X := 1; X := X + 1) \approx X := 2$, *et aussi* $\mathsf{while}\ \overline{\top}\ \mathsf{do}\ \mathsf{skip} \approx \mathsf{while}\ \overline{\top}\ \mathsf{do}\ X := X + 1$. *Si l'on prend la commande qui calcule la factorielle ci-dessous et qu'on change la condition $X \leq 1$ en $X \leq 2$, on obtient une commande qui satisfait toujours les énoncés de correction partielle et terminaison, mais qui n'est pas équivalente à l'originale du fait du contenu de l'adresse R dans l'état résultant.*

4 Sémantique opérationnelle à petits pas

[Winskel, 1993, section 2.6] propose en guise d'exercice de définir une sémantique opérationnelle à petits pas pour chaque catégorie d'expressions. Nous détaillons quelque peu ci-dessous.

Pour les expressions arithmétiques, on pourrait définir une relation entre des paires composées d'une expression et d'un état, mais comme l'état ne change pas, on peut alléger :

Definition 4. *On définit une relation ternaire $\to_1^{\mathbf{A}} \subseteq \mathbf{A} \times \Sigma \times \mathbf{A}$, dont les instances sont notées $a \to_\sigma a'$, comme la plus petite relation close par les règles suivantes, où $\star \in \{+, \times, -\}$:*

$$\frac{}{X \to_\sigma \overline{\sigma(X)}} \qquad \frac{}{\overline{n} \ \overline{\star}\ \overline{m} \to_\sigma \overline{n \star m}}$$

$$\frac{a_1 \to_\sigma a'_1}{a_1 \ \overline{\star}\ a_2 \to_\sigma a'_1 \ \overline{\star}\ a_2} \qquad \frac{a_2 \to_\sigma a'_2}{a_1 \ \overline{\star}\ a_2 \to_\sigma a_1 \ \overline{\star}\ a'_2}$$

Contrairement à ce que propose Winskel, on ne fixe pas ici l'ordre d'évaluation des opérations arithmétiques : on aura ainsi par exemple

$$(\ \overline{1}\ \overline{+}\ \overline{2}\ )\ \overline{+}\ X \to_\sigma\ \overline{3}\ \overline{+}\ X \to_\sigma\ \overline{3}\ \overline{+}\ \overline{\sigma(X)} \to_\sigma \overline{3 + \sigma(X)}$$

en évaluant de gauche à droite ou

$$(\ \overline{1}\ \overline{+}\ \overline{2}\ )\ \overline{+}\ X \to_\sigma (\ \overline{1}\ \overline{+}\ \overline{2}\ )\ \overline{+}\ \overline{\sigma(X)} \to_\sigma\ \overline{3}\ \overline{+}\ \overline{\sigma(X)} \to_\sigma \overline{3 + \sigma(X)}$$

en évaluant de droite à gauche. Dans le cas présent, la stratégie d'évaluation ne peut avoir d'impact sur le résultat ; cela ne serait pas vrai en présence d'effets de bords.

Proposition 3. *Pour tout a, n et σ on $a : a \to_\sigma^* \overline{n}$ ssi $\langle a, \sigma\rangle \to n$.*

Démonstration. Le sens ($\Leftarrow$) se fait facilement par induction sur $\langle a, \sigma\rangle \to n$ ou sur a. Pour le sens ($\Rightarrow$) on procède par induction sur a, en utilisant le fait qu'une réduction de $a_1 \star a_2 \to_\sigma \overline{n}$ contient deux réductions $a_i \to_\sigma \overline{n_i}$ entrelacées (pour chaque $i \in \{1, 2\}$) suivies d'une dernière étape qui calcule l'opération $\star$ à la racine. □

On pourrait montrer, de plus, que toute expression qui n'est pas de la forme $\overline{n}$ peut se réduire par $\to_\sigma$, et que la longueur des réductions à partir d'une expression donnée est bornée. Par les propositions 1 et 3 cela nous donne l'existence d'une forme normale unique pour chaque expression.

Le cas des expressions booléennes est tout à fait analogue. En plus de s'inquiéter de l'ordre d'évaluation des sous-expressions, on se posera la question d'une évaluation paresseuse de la conjonction et de la disjonction, mais en l'absence d'effets (écriture dans l'état, ou non-terminaison) cela n'a pas d'importance.

Pour les commandes, on peut présenter la sémantique à petits pas de multiples façons : voir par exemple les deux variantes de Jean Goubault dans ses notes de cours [2] ou la relation esquissée par Winskel où la dernière étape de calcul est de la forme $\langle c, \sigma\rangle \to_1 \sigma'$ ou encore [Winskel, 1993, Exercise 4.10].

Definition 5. *La relation $\to_1 \subseteq (\mathbf{C} \times \Sigma)^2$ est la plus petite relation close par les règles suivantes :*

$$\frac{}{\langle \mathsf{skip}; c, \sigma\rangle \to_1 \langle c, \sigma\rangle} \qquad \frac{}{\langle (c_1; c_2); c', \sigma\rangle \to_1 \langle c_1; (c_2; c'), \sigma\rangle}$$

$$\frac{}{\langle X := \overline{n}\,; c, \sigma\rangle \to_1 \langle c, \sigma[n/X]\rangle} \qquad \frac{a \to_\sigma a'}{\langle X := a; c, \sigma\rangle \to_1 \langle X := a'; c, \sigma\rangle}$$

$$\frac{}{\langle \mathsf{if}\ \overline{\top}\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c', \sigma\rangle \to_1 \langle c_1; c', \sigma\rangle}$$

$$\frac{}{\langle \mathsf{if}\ \overline{\bot}\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c', \sigma\rangle \to_1 \langle c_2; c', \sigma\rangle}$$

$$\frac{b \to_\sigma b'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c', \sigma\rangle \to_1 \langle \mathsf{if}\ b'\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c', \sigma\rangle}$$

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ c; c', \sigma\rangle \to_1 \langle (\mathsf{if}\ b\ \mathsf{then}\ c; \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}); c', \sigma\rangle}$$

Attention au piège : il ne faudrait surtout pas réduire $\mathsf{while}\ b\ \mathsf{do}\ c; c'$ en $\mathsf{while}\ b'\ \mathsf{do}\ c; c'$ quand $b \to_\sigma b'$, car l'évaluation de b dans les prochaines itérations ne se fera plus forcément sur l'état courant σ.

2. `http://www.lsv.fr/~goubault/CoursProgrammation/prog1_sem1.pdf` § 1.3 et 1.4

L'intuition derrière cette relation est que si $\langle c, \sigma\rangle \to \sigma'$ on pourra réduire $\langle c; \mathsf{skip}, \sigma\rangle \to_1^* \langle \mathsf{skip}, \sigma'\rangle$, et réciproquement : il est nécessaire d'ajouter un skip en séquence après c pour que les règles de réduction s'appliquent. Pour montrer cela il faudra cependant généraliser l'énoncé. Commençons par quelques propriétés plus basiques.

Proposition 4 (Progrès)**.** *Pour tout c, c_k et σ il existe c' et σ' tel que $\langle c; c_k, \sigma\rangle \to_1 \langle c', \sigma'\rangle$.*

Proposition 5. *Pour tout c, c_k, σ, c_r et σ' tel que $\langle c; c_k, \sigma\rangle \to_1 \langle c_r, \sigma'\rangle$, c_r est de l'une des formes suivantes :*

- *soit $c_r \equiv c_k$;*
- *soit c_r est de la forme $c'; c_k$;*
- *soit c_r est de la forme $c'; (c''; c_k)$.*

Ces observations indiquent qu'on gagne à considérer que notre sémantique à petits pas ne travaille pas sur les commandes mais sur une commande de la forme $c; c_k$ où la composante c_k évolue comme une pile. Quand on réduit $\langle c; c_k, \sigma\rangle$ on est en fait en train de calculer c avec une *continuation* c_k qui indique ce qu'il faudra faire ensuite (voir section 7.2 pour une extension exploitant cette notion).

Proposition 6. *Pour tout c, σ et σ' tel que $\langle c, \sigma\rangle \to \sigma'$ on a, pour tout c_k, $\langle c; c_k, \sigma\rangle \to_1^* \langle c_k, \sigma'\rangle$.*

Démonstration. Par induction sur $\langle c, \sigma\rangle \to \sigma'$. Le cas intéressant pour comprendre le rôle de c_k est celui de l'exécution d'une séquence. □

Proposition 7. *Pour tout c, c_k, σ et σ' tel que $\langle c; c_k, \sigma\rangle \to_1^* \langle c_k, \sigma'\rangle$, on a $\langle c, \sigma\rangle \to \sigma'$.*

Idée de preuve. Par induction sur la longueur de la réduction de $\langle c; c_k, \sigma\rangle$. On utilisera quelques propriétés des réductions.

Pour traiter le cas où $c \equiv c_1; c_2$, on observera qu'une réduction $\langle c_1; c_2; c_k, \sigma\rangle \to_1^* \langle c_k, \sigma'\rangle$ est forcément de la forme $\langle c_1; c_2; c_k, \sigma\rangle \to_1^* \langle c_2; c_k, \sigma''\rangle \to_1^* \langle c_k, \sigma'\rangle$ ce qui permet de conclure par hypothèses d'inductions.

Ou encore, pour le cas de l'assignation, on utilisera le fait qu'une réduction $\langle X := a; c_k, \sigma\rangle \to_1^* \langle c_k, \sigma'\rangle$ est forcément de la forme $\langle X := a; c_k, \sigma\rangle \to_1^* \langle X := \overline{n}\,; c_k, \sigma\rangle \to_1 \langle c_k, \sigma[n/X]\rangle$ avec $a \to_\sigma \overline{n}$. □

On a choisi ici de n'utiliser que la sémantique à petits pas pour les sous-expressions, ce qui induit une certaine lourdeur. Alternativement on pourrait utiliser la sémantique à grands pas, ou même dénotationnelle pour les expressions, et ne traiter à petits pas que les commandes : cela ne serait pas une hérésie d'autant que le style à petits pas n'est particulièrement intéressant que pour celles ci. En effet, contrairement au cas des expressions, on observe ici des cas de non-terminaison ; c'est un intérêt important de la sémantique à petits pas par rapport à la sémantique à grands pas.

Exemple 4. *Soit $c \equiv (\mathsf{while}\ \overline{\top}\ \mathsf{do}\ \mathsf{skip}); \mathsf{skip}$. On a $\langle c, \sigma\rangle \to_1^3 \langle c, \sigma\rangle$ pour tout σ.*

La propriété de progrès montrée précédemment indique en fait que la non-terminaison est le seul cas où une commande ne s'exécute pas dans la sémantique à grands pas.

Proposition 8. *Pour tout c et σ, on a (1) ou (2) :*
(1) Il existe σ' tel que $\langle c, \sigma\rangle \to \sigma'$.
(2) Il existe une réduction infinie par $\to_1$ à partir de $\langle c; \mathsf{skip}, \sigma\rangle$.

Avec un peu de travail supplémentaire on pourrait montrer qu'on n'a jamais les deux en même temps. Cela serait immédiat si $\to_1$ était déterministe, ce qu'on peut obtenir en prenant $\to_1^{\mathbf{A}}$ et $\to_1^{\mathbf{B}}$ déterministes.

5 Sémantique dénotationnelle

On suit ici [Winskel, 1993, chapter 5], où l'on interprète directement les différentes catégories grammaticales en des représentations de ce qu'elles calculent :

$$\begin{array}{rcl} [\![a]\!] & : & \Sigma \to \mathbb{Z} \\ [\![b]\!] & : & \Sigma \to \mathbb{B} \\ [\![c]\!] & : & \Sigma \rightharpoonup \Sigma \end{array}$$

Comme attendu, les fonctions sont totales pour les expressions mais partielles pour les commandes.

La définition de la sémantique dénotationnelle des expressions n'est pas très intéressante : on pourrait aussi bien définir $[\![e]\!]$ comme l'unique fonction qui à σ associe l'unique v tel que $\langle e, \sigma\rangle \to v$, et trivialiser ainsi l'adéquation avec la sémantique à grands pas.

On notera que Winskel définit chaque sémantique d'abord dans les relations, avant de montrer que ce sont en fait des fonctions qui sont définies. On peut très bien s'en passer, ce que je fais ci-dessous en détaillant le cas des commandes.

Definition 6. *On équipe $\Sigma \to \Sigma$ de l'ordre $\leq$ défini comme suit : $f \leq g$ si $\mathsf{dom}(f) \subseteq \mathsf{dom}(g)$ et $g|_{\mathsf{dom}(f)} = f$. On note $f_\perp$ la fonction définie nulle part, plus petit élément de $\Sigma \rightharpoonup \Sigma$ pour $\leq$. Toute suite croissante $f_0 \leq \ldots \leq f_i \leq f_{i+1} \leq \ldots$ admet une borne supérieure $\sup_{i\in\mathbb{N}} f_i$ qui est l'unique fonction définie sur $\cup_{i\in\mathbb{N}}\mathsf{dom}(f_i)$ et coïncidant avec chaque f_i sur son domaine de définition.*

Definition 7. *La sémantique $[\![c]\!]$ d'une commande c se définit par induction sur sa structure :*

$$\begin{array}{rcl} [\![\mathsf{skip}]\!] & = & \sigma \mapsto \sigma \\ [\![X := a]\!] & = & \sigma \mapsto \sigma[[\![a]\!](\sigma)/X] \\ [\![c_1; c_2]\!] & = & [\![c_2]\!] \circ [\![c_1]\!] \\ [\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] & = & \sigma \mapsto \begin{cases} [\![c_1]\!](\sigma) \ \textit{si}\ [\![b]\!](\sigma) = \top \\ [\![c_2]\!](\sigma) \ \textit{si}\ [\![b]\!](\sigma) = \perp \end{cases} \end{array}$$

$$[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] = \sup_{i\in\mathbb{N}} F_w^i(f_\perp)$$

$$F_w(f) = \sigma \mapsto \begin{cases} \sigma \ si\ [\![b]\!](\sigma) = \perp \\ f \circ [\![c]\!] \ si\ [\![b]\!](\sigma) = \top \end{cases}$$

où l'on note $w \equiv \mathsf{while}\ b\ \mathsf{do}\ c$ *dans les deux dernières lignes.*

Exemple 5. *Soit c la commande calculant la factorielle dans l'exemple 2. On a* $\mathsf{dom}([\![c]\!]) = \{\sigma \mid \sigma(X) \geq 0\}$ *et* $[\![c]\!](\sigma) = \sigma[\sigma(X)!/R, 0/X]$. *On pourrait détailler l'exemple en donnant les* $F_w^i(f_\perp)$ *où w est la boucle de c.*

Proposition 9. *Pour tous c,* σ *et* σ' *on a* $[\![c]\!](\sigma) = \sigma'$ *ssi* $\langle c, \sigma\rangle \to \sigma'$.

Indications de preuve. Pour ($\Leftarrow$) on procède simplement par induction sur l'exécution. Pour le cas d'une boucle w on utilisera une induction sur $\langle w, \sigma\rangle \to \sigma'$ pour montrer plus précisément qu'il existe un i tel que $F_w^i(f_\perp)(\sigma) = \sigma'$. Cela suffit en effet pour conclure $[\![w]\!](\sigma) = \sigma'$.

Pour ($\Rightarrow$) on raisonne par induction sur c. Dans le cas d'une boucle w on utilise simplement le fait que si $[\![w]\!](\sigma) = \sigma'$ alors il existe i tel que $F_w^i(\sigma) = \sigma'$ et on raisonne par induction sur i pour établir $\langle w, \sigma\rangle \to \sigma'$. □

Autrement dit, notre sémantique est *pleinement abstraite* : $c \approx c'$ ssi $[\![c]\!] = [\![c']\!]$. Mieux, on peut prouver la réciproque de la proposition 2.

Proposition 10. *Soient c et c' des commandes. Si* $c \approx c'$ *alors, pour tout contexte C, on a* $\langle C[c], \mathbf{0}\rangle$ *termine ssi* $\langle C[c'], \mathbf{0}\rangle$ *termine.*

Démonstration. Il suffit d'observer que $[\![C[c]]\!] = [\![C[c']]\!]$ puisque notre sémantique est compositionnelle. □

On remarque qu'on n'a pas eu besoin de parler de point fixe, monotonicité ou continuité, pour définir notre sémantique et montrer quelques résultats intéressants. Il est néanmoins utile d'aller plus loin : pour savoir que notre borne supérieure est un point fixe, et pour pouvoir obtenir des constructions similaires sur des espaces plus complexes (cf. section 7.3). Je reprends ci-dessous la notion de cpo de [Winskel, 1993, section 5.4] en forçant la présence d'un élément minimal, qui est un peu plus restrictive que la notion de dcpo du cours de Jean Goubault.

Definition 8. *Un ordre partiel* $(D, \sqsubseteq)$ *est un cpo (ordre partiel complet) si :*
- *D a un plus petit élément* $\perp_D$ *;*
- *toute chaîne* $d_0 \sqsubseteq \ldots \sqsubseteq d_i \sqsubseteq d_{i+1} \sqsubseteq \ldots$ *admet une borne supérieure* $\sqcup_i d_i$.

Definition 9. *Soient D et E des cpos. Une fonction* $f : D \to E$ *est* continue *si elle est croissante et, pour toute chaîne croissante* $d_0 \sqsubseteq \ldots \sqsubseteq d_i \sqsubseteq d_{i+1} \sqsubseteq \ldots$, $\sqcup_i f^i(d_i) = f(\sqcup_i d_i)$.

Proposition 11 (Theorem 5.11 dans Winskel [1993]). *Soit D un cpo et $f : D \to D$ continue. On pose $\mathsf{lfp}(f) := \sqcup_i f^i(\bot_D)$. Alors $\mathsf{lfp}(f)$ est un point fixe de f, et le plus petit des points pre-fixes, i.e. pour tout x tel que $f(x) \leq x$, $\mathsf{lfp}(f) \leq x$.*

Il est clair que $(\Sigma \rightharpoonup \Sigma, \leq)$ forme un cpo. Pour utiliser le résultat précédent on peut de plus vérifier que F_w est continue.

Exemple 6. *Soit $w \equiv \mathsf{while}\ b\ \mathsf{do}\ c$. Comme $[\![w]\!]$ est un point fixe de F_w, on a $[\![\mathsf{if}\ b\ \mathsf{then}\ (c; w)\ \mathsf{else}\ \mathsf{skip}]\!] = [\![w]\!]$.*

Les amateurs de points fixes peuvent aussi redémontrer la proposition 9 en utilisant la caractérisation de $[\![w]\!]$ comme le plus petit des points pre-fixes : pour la direction ($\Rightarrow$), dans le cas d'une boucle w, il suffit de montrer que la fonction de graphe $\{(\sigma, \sigma') \mid \langle w, \sigma\rangle \to \sigma'\}$ est un point pre-fixe de F_w.

Il est moins intéressant de parler du théorème de Knaster-Tarski sur les points fixes de fonctions croissantes dans un treillis complet (cf. [Winskel, 1993, section 5.5]) car ce théorème s'applique moins naturellement ici. En effet il faudrait travailler avec des relations plutôt que des fonctions $\Sigma \rightharpoonup \Sigma$ pour avoir un treillis complet.

6 Sémantique axiomatique

C'est une sémantique aussi, on peut en parler, mais sans trop déborder sur d'autres leçons. C'est traité dans [Winskel, 1993, chap. 6 & 7], on pourra simplifier les définitions pour se rapprocher de la logique du premier ordre en fusionnant état et valuations. La section 7.5 propose une sémantique dénotationnelle à peine modifiée permettant de voir une commande comme un transformateur de prédicats, en fait un calcul de précondition, faisant ainsi le lien avec la sémantique axiomatique.

7 Extensions

On propose ici quelques extensions qui ont pour but d'illustrer certains points qui pourraient passer inaperçus dans Imp, et d'élargir la discussion vers des langages concrets.

7.1 Effets de bords dans les expressions

Considérons l'ajout de la construction $X{+}{+}$ aux expressions arithmétiques :

$$a ::= \ldots \mid X{+}{+}$$

La sémantique souhaitée est comme en C : l'expression prend la valeur initiale de X mais incrémente celle ci en mémoire. Cette modification demande de changer les sémantiques à grands pas des expressions : on aurait maintenant

$\langle e, \sigma\rangle \to \langle v, \sigma'\rangle$ où e est une expression arithmétique (resp. booléenne) et v une valeur de $\mathbb{Z}$ (resp. $\mathbb{B}$). La règle pour l'incrément serait

$$\frac{}{\langle X{+}{+}, \sigma\rangle \to \langle \sigma(X), \sigma[\sigma(X)+1/X]\rangle}$$

mais les autres règles devraient être adaptées pour chaîner les modifications d'état et préciser l'ordre d'évaluation, par exemple

$$\frac{\langle a_1, \sigma\rangle \to \langle n_1, \sigma'\rangle \quad \langle a_2, \sigma'\rangle \to \langle n_2, \sigma''\rangle}{\langle a_1 \mathbin{\overline{\star}} a_2, \sigma\rangle \to \langle n_1 \star n_2, \sigma''\rangle}$$

Dans ce cadre, la différence entre connecteurs booléens paresseux ou stricts prendrait aussi son sens : les connecteurs stricts de Winskel [1993] évaluent leurs deux sous-expressions quels que soient leurs résultats, mais des connecteurs paresseux pourraient s'en passer, par exemple avec la règle suivante :

$$\frac{\langle b_1, \sigma\rangle \to \langle \top, \sigma'\rangle}{\langle b_1 \vee b_2, \sigma\rangle \to \langle \top, \sigma'\rangle}$$

7.2 Commandes break et continue

Un intérêt de la sémantique à petits pas vue plus haut, qui donne accès à une notion de continuation, est qu'on peut définir la sémantique d'opérateurs de contrôle comme break et continue. Pour cette extension, nous supposerons que chaque construction while dans la commande de départ est annotée par un identifiant unique, ce qu'on écrit $(\mathsf{while}\ b\ \mathsf{do}\ c)^\alpha$. On étend la syntaxe des commandes par les deux opérateurs, décorés aux aussi par un identifiant de boucle :

$$c ::= \ldots \mid \mathsf{break}^\alpha \mid \mathsf{continue}^\alpha$$

La sémantique à petits pas est étendue par les nouvelles règles suivantes :

$$\frac{}{\langle \mathsf{break}^\alpha; c_1; \ldots; c_n; w^\alpha; c', \sigma\rangle \to_1 \langle c', \sigma\rangle}$$

$$\frac{}{\langle \mathsf{continue}^\alpha; c_1; \ldots; c_n; w^\alpha; c', \sigma\rangle \to_1 \langle w^\alpha; c', \sigma\rangle}$$

Dans les deux règles, les c_i sont des commandes non annotées par α, ce qui est garanti par l'unicité des annotations.

Exemple 7. *La commande*

$$\begin{array}{l}(\mathsf{while}\ \overline{\top}\ \mathsf{do}\\ \quad R := R + X; X := X - 1; \mathsf{if}\ X = 0\ \mathsf{then}\ \mathsf{break}^\alpha\ \mathsf{else}\ (\mathsf{continue}^\alpha; R := 0))^\alpha\end{array}$$

termine si le contenu initial de X est strictement supérieur à 0, et ajoute à R la somme des entiers de 1 au contenu initial X. La sous-commande $R := 0$ n'est jamais exécutée.

La sémantique obtenue ne peut être comparée à une sémantique dénotationnelle ou à grands pas, qui seraient nettement plus lourdes à étendre. Mais elle est conforme à l'esprit de ces opérateurs dans les langages de programmation courants : C, Python, etc. Dans plusieurs langages (dont Javascript, Java, Rust, ou encore Perl où continue s'appelle next) on peut même utiliser des identifiants de boucle pour utiliser ces opérateurs vis-à-vis d'autres boucles que la boucle courante, i.e. la plus profonde, ce qui correspond aux identifiants que nous avons introduit ici.

7.3 Procédures mutuellement récursives

On définit un *programme* de taille n comme un n-uplet de commandes $\langle c_1, \ldots, c_n \rangle$. La commande c_i est vue comme la définition d'une procédure qui sera simplement indentifiée par l'index i. Les commandes pourront appeler ces procédures via l'extension suivante de la syntaxe des commandes :

$$c ::= \ldots \mid \mathsf{call}_i$$

En particulier, les procédures sont ainsi mutuellement récursives. On dira qu'une commande est n-définie quand tous les appels appels call_i qu'elle contient satisfont $1 \leq i \leq n$. On imposera que les procédures d'un programme de taille n soient n-définies.

Étant donné un programme $p = \langle c_1, \ldots, c_n \rangle$ on peut définir la sémantique à grands pas d'une commande n-définie en ajoutant une unique règle au système :

$$\frac{\langle c_i, \sigma \rangle \to \sigma'}{\langle \mathsf{call}_i, \sigma \rangle \to \sigma'}$$

Pour la sémantique dénotationnelle, il faut traiter cette récursion mutuelle. Un programme de taille n s'interprète comme un objet P du cpo produit $(\Sigma \rightharpoonup \Sigma)^n$. Pour chaque P on peut définir une sémantique $[\![c]\!]^P$ des commandes, en étendant la définition 7 par $[\![\mathsf{call}_i]\!]^P = P_i$. On définit enfin $F_p := P \mapsto \langle [\![c_1]\!]^P, \ldots, [\![c_n]\!]^P \rangle$, pour poser $[\![p]\!] = \sqcup_i F_p^i(\bot)$.

Proposition 12. *$\langle c, \sigma \rangle \to \sigma'$ entraîne $[\![c]\!]^{[\![p]\!]}(\sigma) = \sigma'$.*

Indications de preuve. En fait F_p est continue et $[\![p]\!] = \mathsf{lfp}(F_p)$, donc $[\![c_i]\!]^{[\![p]\!]} = [\![p]\!]_i$. Cela permet d'étendre la preuve précédente, toujours par induction sur l'exécution à grands pas. □

Proposition 13. *$[\![c]\!]^{[\![p]\!]}(\sigma) = \sigma'$ entraîne $\langle c, \sigma \rangle \to \sigma'$.*

Indications de preuve. Si l'on veut reprendre l'induction sur c du cas de Imp, pour le nouveau cas d'un call_i il nous faudra montrer que si $[\![p]\!]_i(\sigma) = \sigma'$ alors $\langle c_i, \sigma \rangle \to \sigma'$. Cela revient à montrer que $[\![p]\!]_i \sqsubseteq \langle \ldots, \{(\sigma, \sigma') \mid (c_i, \sigma) \to \sigma'\}, \ldots \rangle$ ce qu'on obtient en montrant que le membre droit x est un point pre-fixe de F_p. La preuve commencera par établir par induction sur c que $[\![c]\!]^x(\sigma) = \sigma'$ entraîne $\langle c, \sigma \rangle \to \sigma'$, où le cas du call est désormais trivial. On aura donc $[\![p]\!] \sqsubseteq x$, d'où l'on déduira $[\![c]\!]^{[\![p]\!]} \sqsubseteq [\![c]\!]^x$, ce qui permet de conclure. □

Références

Glynn Winskel. *The formal semantics of programming languages : an introduction*. MIT press, 1993.