Sémantique des langages de programmation deuxième partie: sémantique dénotationelle

Yves Bertot

Février 2008

1 Qu'est-ce que la sémantique dénotationnelle

Dans la leçon précédente nous avons décrit la sémantique d'un langage de programmation en fournissant un prédicat qui mettait en relation les programmes, les données en entrée et les données en sortie. Les propriétés de ce prédicat étaient décrites par un jeu de règles de raisonnement.

Nous voulons maintenant prendre un point de vue plus directement mathématique: chaque programme représente une fonction mathématique partielle d'un ensemble vers un autre. L'objectif de ce chapître va être de préciser les ensembles de départ et d'arrivée des fonctions considérées et les procédés utilisés pour décrire les fonctions obtenues: certains de ces procédés font appel à des notions mathématiques assez complexes.

1.1 Définir des fonctions récursives structurelles

Quand on a défini un ensemble d'arbres, il est facile de définir des fonctions qui calculent récursivement sur ces arbres. Chaque arbre est de la forme $f(t_1, \ldots, t_k)$ et une fonction récursive structurelle est une fonction qui calcule sa valeur sur cet arbre en utilisant potentiellement les valeurs de la *même fonction* sur les arbres t_1, $\ldots$, t_k. Ceci généralise aux arbres la notion de fonction récursive primitive pour les entiers naturels (voir aussi les suites récurrentes simples).

Les fonctions récursives sont définies en donnant leur comportement pour chacun des cas possibles d'arbres, en faisant des appels sur les sous arbres directs. Par exemple, si l'on cherche à mesurer le nombre de nœuds d'un programme vu comme un arbre on définira la fonction *size* qui vérifie les propriétés suivantes:

$$
\begin{aligned}
size(while(e, I)) &= 1 + size(e) + size(I) \\
size(assign(x, e)) &= 1 + size(x) + size(e) \\
size(sequence(I_1, I_2)) &= 1 + size(I_1) + size(I_2) \\
size(greater(e_1, e_2)) &= 1 + size(e_1) + size(e_2) \\
size(plus(e_1, e_2)) &= 1 + size(e_1) + size(e_2) \\
size(n) &= 1 \\
size(x) &= 1
\end{aligned}
$$

Exercices

1. *donner la définition récursive structurelle d'une fonction $variables_1$ qui retourne la liste des variables apparaissant dans une instruction, en utilisant une fonction* `append` *pour réunir deux listes (sans se préoccuper des duplications).*

2. *donner la définition d'une fonction $variables_2$ qui prend deux arguments, le premier étant une liste de variables, le second une instruction, et retourne la liste contenant toute les variables présentes dans la première liste et toutes les variables présentes dans l'instruction. Cette fonction sera récursive structurelle par rapport à l'instruction et ne devra pas utiliser la fonction* `append`.

3. *La fonction* `append` *a une complexité en temps et en mémoire proportionnelle à la taille de son premier argument. Si vous avez correctement répondu aux deux exercices précédents, vous devez être capable de construire une suite d'instructions p_i tel que la taille du programme i soit $O(i)$, le temps de calcul soit environ $variables_1(p_i)$ soit $O(i^2)$, et le coût de $variables_2(p_i)$ soit $O(i)$.*

4. *Ecrire les fonctions size, $variables_1$ et $variables_2$ en Coq ou en Ocaml.*

1.2 La sémantique dénotationelle des expressions

1.2.1 Notation pour l'utilisation d'un environnement

Dans ce chapitre, nous utilisons les variables de nom σ pour représenter les environnements. Ici, je choisis cette convention parce que les environnements sont aussi parfois appelés des états (en anglais *state*). Cette convention est aussi parfois utilisée en théorie du λ-calcul, de l'unification, et de la réécriture, car on pense alors à des *substitutions*, qui sont également des fonctions associant des variables à des valeurs.

Dans la suite nous noterons *state* le type des environnements.

La recherche de la valeur d'une variable dans un environnement correspond à une fonction récursive structurelle, mais pas totale, car la variable cherchée peut ne pas apparaître dans l'environnement.

Ceci se représente par le fait que l'on peut définir une fonction $flookup$ (le préfixe $flookup$ est utilisé pour indiquer qu'il s'agit d'une fonction) telle que la valeur de la variable v dans l'environnement σ soit obtenue par l'expression $flookup(\sigma, v)$. Cette fonction est définie par les équations suivantes:

$$\begin{array}{lcll} flookup((x,n)\cdot\sigma, x) & = & n & \\ flookup((x,n)\cdot\sigma, y) & = & flookup(\sigma, y) & (x \neq y) \end{array}$$

Notez qu'il n'y a pas d'équation donnant la valeur d'une variable dans l'environnement vide. C'est pour cette raison que la fonction est partielle.

1.2.2 Mise à jour des environnements

L'opération de mise à jour d'un environnement peut aussi être décrite par une fonction récursive structurelle. Nous noterons cette fonction $\sigma[x \leftarrow n]$ et nous la définirons par les équations suivantes:

$$\begin{array}{lcll} ((x,n)\cdot\sigma)[x \leftarrow n'] & = & (x,n')\cdot\sigma & \\ ((x,n)\cdot\sigma)[y \leftarrow n'] & = & (x,n)\cdot(\sigma[y \leftarrow n']) & (x \neq y) \end{array}$$

Ici encore, il n'y a pas d'équation pour le cas où l'on voudrait mettre à jour l'environnement vide, ce qui traduit le fait que cette fonction de mise à jour est également partielle.

La fonction de recherche de valeur et la fonction de mise à jour sont sont cohérentes. En particulier elles vérifient les équations suivantes:

$$\begin{array}{rcl} flookup(\sigma[x \leftarrow n], x) & = & n \\ flookup(x \neq y \Rightarrow \sigma[x \leftarrow n], y) & = & \sigma(y) \end{array}$$

Exercices

5. *Démontrer les égalités suivantes*

$$\sigma[x \leftarrow n][x \leftarrow m] = \sigma[x \leftarrow m]$$

$$\sigma[x \leftarrow n][y \leftarrow m] = \sigma[y \leftarrow m][x \leftarrow n] \quad (x \neq y)$$

1.2.3 Notations pour les fonctions

Nous serons souvent amenés à considérer des fonctions définies directement par des phrases de la forme « la fonction qui à x associe la valeur $expr$ », où x apparait dans l'expression $expr$. Pour ces fonctions, nous utiliserons la notation $\lambda x.expr$. Par exemple, la fonction qui à x associe $x + 1$ est écrite de la façon suivante:

$$\lambda x.\ x + 1$$

1.2.4 Sémantique des expressions

Nous décomposons la sémantique des expressions en deux parties, l'une pour les expressions booléennes et l'autre pour les expressions entières. Ces deux parties sont décrites à l'aide de deux fonctions que nous noterons pour l'instant $\mathcal{A}$ et $\mathcal{B}$.

Donnons d'abord la sémantique des expressions arithmétiques:

$$\begin{aligned} \mathcal{A}[\![n]\!] &= \lambda\sigma.n \\ \mathcal{A}[\![x]\!] &= \lambda\sigma.flookup(\sigma, x) \\ \mathcal{A}[\![plus(e_1, e_2)]\!] &= \lambda\sigma.\mathcal{A}[\![e_1]\!](\sigma) + [\![e_2]\!](\sigma) \end{aligned}$$

La notation $[\![\cdot]\!]$ est usuelle en sémantique dénotationelle. la fonction $\mathcal{A}$ a le type suivant:

$$exp \to (state) \to \mathbb{Z}.$$

On peut la voir comme une fonction à un ou deux arguments. Vue comme une fonction à un argument, elle est définie pour toutes les expressions, mais sa valeur est une fonction partielle des environnements vers les valeurs entières, parce qu'elle repose sur la fonction $flookup$ qui est une fonction partielle. Lorsque l'on calcule la valeur $\mathcal{A}[\![x]\!]$, il est possible que la valeur $\sigma(x)$ ne soit pas définie. De manière générale, la fonction $\mathcal{A}[\![exp]\!]$ est définie pour tous les environnements qui sont définis pour toutes les variables apparaissant dans x.

Vue comme une fonction des expressions vers les fonctions partielles des environnements vers les valeurs, la fonction $\mathcal{A}$ n'est pas partielle mais totale.

Pour les expressions booléennes, le calcul est similaire. La définition utilise une condition, comme la définition d'une fonction par morceaux.

$$\begin{aligned} \mathcal{B}[\![greater(e_1, e_2)]\!] = \lambda\sigma. \quad & \text{si } \mathcal{A}[\![e_1]\!](\sigma) \text{ et } \mathcal{A}[\![e_2]\!] \text{ sont définies,} \\ & \text{si } \mathcal{A}[\![e_1]\!](\sigma) > \mathcal{A}[\![e_2]\!](\sigma) \text{ alors } true \text{ sinon } false \end{aligned}$$

La fonction $\mathcal{B}$ a le type $exp \to (state) \to \mathbb{B}$. Comme la fonction $\mathcal{A}$, elle est totale en son premier argument, mais la valeur retournée est une fonction partielle sur les environnements. Si l'une des valeurs $\mathcal{A}[\![e_1]\!](\sigma)$ ou $\mathcal{A}[\![e_2]\!](\sigma)$ n'est pas définie, alors $\mathcal{B}[\![greater(e_1, e_2)]\!](\sigma)$ n'est pas défini.

1.3 La sémantique dénotationelle des instructions

1.3.1 Décrire des fonctions partielles

Raisonner sur des fonctions partielles est difficile, car on est toujours ennuyé de parler d'une expression qui n'est peut-être pas définie.

Nous avons décrit les environnements comme des fonctions partielles sur les variables. La présentation aurait été moins complexe si nous avions décidé de n'utiliser que des environnements définis partout, par exemple en les étendant

avec une valeur par défaut. Dans ce cas, les fonctions de dénotation pour les expressions auraient été totales également.

Pour les instructions, le problème se pose différemment. Il est naturel de considérer que le sens d'une instruction est une fonction qui prend un état de la machine (c'est à dire un environnement) et retourne un nouvel état de la machine. Néanmoins, il existe des instructions dont l'exécution ne termine pas et leur valeur est alors indéfinie, même si l'environnement donné en argument est bien défini pour toutes les variables présentes dans le programme. Prenons par exemple l'instruction

$$while(greater(1,0), assign(x,x))$$

la sémantique de cette instruction est une fonction qui n'est définie pour aucun environnement, parce que le programme ne termine pas.

Pour manipuler des fonctions totales plutôt que des fonctions partielles, nous nous donnons une valeur $\perp$ qui va servir à représenter la valeur indéfinie, et nous allons ajouter cette valeur à certains des ensembles de valeurs que nous considérons. Pour un ensemble A donné, nous prolongerons les fonctions partielles à valeur dans A en des fonctions totales à valeur dans $A \cup \{\perp\}$. Nous noterons $A_\perp$ l'ensemble$A \cup \{\perp\}$.

Pour tenir compte du fait que les interprétations de certaines instructions peuvent être des fonctions partielles, nous considérons des fonctions du type $state \to state_\perp$. Les fonctions de traitement des environnements $flookup$ et $\cdot[\cdot \to \cdot]$ apparaissent également comme des fonctions partielles. Leurs types sont respectivement:

$$\begin{aligned} flookup &: state \to V \to \mathbb{Z}_\perp \\ \cdot[\cdot \leftarrow \cdot] &: state \to V \to \mathbb{Z} \to state_\perp \end{aligned}$$

Pour l'évaluation des expressions arithmétiques et booléennes, nous devons également tenir compte de l'éventualité où les expressions considérées n'ont pas de valeur. Les fonctions prennent les types suivants:

$$\mathcal{A} : exp \to state \to \mathbb{Z}_\perp$$

$$\mathcal{B} : exp \to state \to \mathbb{B}_\perp$$

Il est alors aussi nécessaire de changer la sémantique associée aux différents opérateurs pour tenir compte du fait que certaines valeurs sont indéterminées. La façon raisonable d'étendre les différentes fonctions est de considérer que la valeur retournée est indéterminée dès que l'une des valeurs en entrée est indéterminée. Pour la sémantique de *greater* nous devrons faire de la façon suivante:

$$
\begin{array}{rcll}
\mathcal{A}[\![n]\!] & = & \lambda\sigma. & n \\
\mathcal{A}[\![v]\!] & = & \lambda\sigma. & flookup(\sigma, v) \\
\mathcal{A}[\![e_1 + e_2]\!] & = & \lambda\sigma. & \text{si } \mathcal{A}[\![e_1]\!](\sigma) = \bot \text{ ou } \mathcal{A}[\![e_2]\!](\sigma) = \bot \text{ alors } \bot \\
 & & & \text{sinon}, \mathcal{A}[\![e_1]\!](\sigma) + \mathcal{A}[\![e_2]\!](\sigma) \\
\mathcal{B}[\![greater(e_1, e_2)]\!] & = & \lambda\sigma. & \text{si } \mathcal{A}[\![e_1]\!](\sigma) = \bot \text{ ou } \mathcal{A}[\![e_2]\!](\sigma) = \bot \text{ alors } \bot \\
 & & & \text{sinon, si } \mathcal{A}[\![e_1]\!](\sigma) > \mathcal{A}[\![e_2]\!](\sigma) \text{ alors } true \\
 & & & \text{sinon } false
\end{array}
$$

1.3.2 Sémantique des instructions

Nous allons maintenant décrire une fonction qui associe un sens aux instructions, la fonction $\mathcal{I}$ qui a le type suivant:

$$\mathcal{I} : instr \to state \to state_\bot$$

$$
\begin{array}{rcl}
\mathcal{I}[\![assign(x, e)]\!] & = & \lambda\sigma.\text{si } \mathcal{A}[\![e]\!](\sigma) = \bot \text{ alors } \bot \\
 & & \text{sinon } \sigma[x \leftarrow \mathcal{A}[\![e]\!](\sigma)] \\
\mathcal{I}[\![sequence(I_1, I_2)]\!] & = & \lambda\sigma.\text{si } \mathcal{I}[\![I_1]\!](\sigma) = \bot \text{ alors } \bot \\
 & & \text{sinon } \mathcal{I}[\![I_2]\!](\mathcal{I}[\![I_1]\!](\sigma)) \\
\mathcal{I}[\![while(e, I)]\!] & = & \phi(\mathcal{A}[\![e]\!], \mathcal{I}[\![I]\!])
\end{array}
$$

Dans cette expression, nous avons laissé une inconnue, la fonction ϕ. Nous nous contenterons de dire que la fonction ϕ est la fonction vérifiant la propriété suivante:

$$
\begin{array}{rcll}
\phi(t, f) & = & \lambda\sigma. & \text{si } t(\sigma) = \bot \text{ alors } \bot \\
 & & & \text{sinon} \quad \text{si} \quad t(\sigma) = true \text{ alors} \\
 & & & \qquad \text{si } f(\sigma) = \bot \text{ alors } \bot \\
 & & & \qquad \text{sinon } \phi(t, f)(f(\sigma))) \\
 & & & \quad \text{sinon } \sigma
\end{array}
$$

La question importante est que cette propriété ne semble pas vraiment être une définition, puisque $\phi(t, f)$ apparait des deux cotés. La fonction $\mathcal{I}$ ne peut pas non plus être écrite comme une fonction récursive structurelle, la fonction ϕ est-elle vraiment définie par cette propriété? On peut répondre par l'affirmative: c'est une conséquence d'un théorème connu sous le nom de *théorème du point fixe*.

1.3.3 Le théorème du point fixe

Pour montrer l'existence de la fonction ϕ nous allons utiliser quelques notions simples qui rappellent la topologie. On sait que toute fonction contractante sur un ensemble compact admet un point fixe. Ici nous allons également décrire la fonction ϕ comme le point fixe d'une fonction ayant des propriétés topologiques. Mais les notions que nous utiliserons pour définir les fonctions continues seront beaucoup plus simples qu'en topologie générale ou en analyse réelle ou complexe.

Définition 1 *un* ordre partiel *est un ensemble muni d'une relation réflexive, antisymétrique et transitive.*

Dans la suite nous noterons D l'ensemble et $\sqsubseteq$ la relation d'ordre.

Définition 2 *Un ordre partiel est dit* complet *si toute suite infinie croissante admet de plus petits majorants.*

Les suites croissantes jouent le rôle des suites de Cauchy, et le plus petit majorant joue le rôle de limite, surtout que le plus petit majorant est unique.

Théorème 1 *Si une suite infinie croissante admet un plus petit majorant, alors il est unique.*

Démonstration. Soit u_n une suite infinie croissante et soient m_1 et m_2 deux plus petits majorants de u_n. Comme m_1 est un plus petit majorant de u_n et comme m_2 est un majorant, on a $m_1 \sqsubseteq m_2$, en retournant le raisonnement on a $m_2 \sqsubseteq m_1$, en utilisant l'antisymétrie de l'ordre on obtient $m_2 = m_1$.

Exercices

6. *Montrer que l'ensemble des parties d'un ensemble quelconque, muni de l'inclusion forme un ordre partiel complet.*

Nous noterons $\bigsqcup_{i \in N} d_i$ ou même $\bigsqcup d_i$ le plus petit majorant de la suite $d_i (i \in N)$.

Définition 3 *Une fonction f entre deux ordres partiels est* monotone *si*

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y).$$

Cette définition de la monotonie est assez naturelle et rejoint la notion de monotonie sur les ensembles de nombres.

Exercices

7. *Soient X et Y deux ensembles quelconques et f une fonction partielle de X dans Y. On définit $\hat{f}$ la fonction de l'ensemble des parties de X vers l'ensemble des parties de Y telle que:*

   $$\hat{f}(P) = \{y \in Y | y = f(x) \wedge x \in P\}$$

   Montrer que $\hat{f}$ est monotone. En mathématiques, on ne fait habituellement pas de différence entre f et $\hat{f}$ et la deuxième écriture n'est jamais utilisée.

Définition 4 *Une fonction monotone est* continue *si sa valeur sur le plus petit majorant d'une suite coincide avec le plus petit des majorants sur les valeurs prises pour chaque élément de la suite:*

$$f(\bigsqcup d_i) = \bigsqcup f(d_i)$$

L'analogie avec les fonction continues sur les nombres réels est maintenue: les fonction continues préservent les limites.

Théorème 2 *Toute fonction continue dans un ordre partiel complet avec un élément minimal admet un point fixe.*

Démonstration. Notons $\perp$ l'élément minimal de notre ordre partiel complet. Considérons la suite d_i telle que:

$$d_i = f^i(\perp)$$

On a $\perp \sqsubseteq d_1$ par construction, puisque $\perp$ est élément minimal. On prouvera aisément par récurrence sur i que $d_i \sqsubseteq d_{i+1}$. Cette suite est donc croissante et admet donc un plus petit majorant d.

$$\begin{aligned} f(d) &= f(\bigsqcup f^i(\perp)) \\ &= \bigsqcup f^{i+1}(\perp) \\ &= (\bigsqcup f^{i+1}(\perp)) \sqcup \{\perp\} \\ &= \bigsqcup f^i(\perp) \\ &= d \end{aligned}$$

Le pas délicat de cette démonstration est à la troisième égalité. L'opération effectuée à cette étape montre que l'on peut toujours déplacer les indices d'une suite sans changer son plus petit majorant (sa limite), en ajoutant $\perp$ comme premier élément de la suite. C.Q.F.D.

Théorème 3 *Si f est une fonction continue dans un ordre partiel avec élément minimal $\perp$, alors $\bigsqcup f^i(\perp)$ est le plus petit point fixe de f.*

Démonstration. Reprenons la démonstration précédente et ses notations. Soit d' un point-fixe quelconque de f. Par définition de $\perp$, on a $\perp \sqsubseteq d'$. Par récurrence, si $f^n(\perp) \sqsubseteq d'$ alors on a $f^{n+1}(\perp) \sqsubseteq f(d') = d'$. Le point-fixe d' de f est donc un majorant de la suite $f^i(\perp)$. Comme d en est le plus petit majorant, on a bien $d \sqsubseteq d'$

Les deux théorèmes ci-dessus mis ensemble constituent ce que nous appellerons le théorème du point fixe.

1.3.4 Utilisation pratique

L'ensemble $state_\perp$, muni de l'ordre $\sqsubseteq$, réflexif, antisymétrique et transitif et tel que

- $\forall \sigma, \sigma'. \quad \sigma \sqsubseteq \sigma' \wedge \sigma \neq \sigma' \Rightarrow \sigma = \perp$,
- $\forall \sigma. \quad \perp \sqsubseteq \sigma$,

est un ordre partiel complet avec élément minimal. Toute suite croissante vérifie soit la propriété

$$\forall i. d_i = \bot$$

soit la propriété

$$\exists i.\ d_i \neq \bot \wedge \forall j.(j < i \Rightarrow d_j = \bot) \wedge (j \geq i \Rightarrow d_j = d_i).$$

Dans le premier cas, le plus petit majorant est $\bot$, dans le deuxième cas c'est d_i.

On peut faire la même construction avec les valeurs booléennes.

L'ensemble des fonctions de *state* dans $state_\bot$, muni de l'ordre $\sqsubseteq$ défini par:

$$f \sqsubseteq g \Leftrightarrow (\forall x. f(x) \sqsubseteq g(x))$$

est un ordre partiel complet avec élément minimal, la fonction $\lambda\sigma.\bot$. Nous allons maintenant nous intéresser particulièrement à cet ordre partiel complet, que nous noterons D_I.

Théorème 4 *Si $f \in D_I$, alors la fonction*

$$comp_f = \lambda g \in D_I.\lambda\sigma.\text{ si } f(\sigma) = \bot \text{ alors } \bot \text{ sinon } g(f(\sigma))$$

est croissante de type $D_I \to D_I$.

Démonstration. soient g_1, g_2 deux fonctions de D_I telles que $g_1 \sqsubseteq g_2$, c'est-à-dire $forall x, g_1(x) \sqsubseteq g_2(x)$. Il faut montrer que $comp_f(g_1) \sqsubseteq comp_f(g_2)$. Pour cela il suffit de montrer

$$\forall\sigma, comp_f(g_1)(\sigma) \sqsubseteq comp_f(g_2)(\sigma).$$

Fixons σ, si $f(\sigma) = \bot$, alors $comp_f(g_1)(\sigma) = comp_f(g_2) = \bot$).
Si $f(\sigma) \neq \bot$ alors $comp_f(g_1)(\sigma) = g_1(f(\sigma))$ et $comp_f(g_2)(\sigma) = g_2(f(\sigma))$. Puisque, $g_1 \sqsubseteq g_2$, on a $g_1(f(\sigma)) = g_2(f(\sigma))$, donc $comp_f(g_1)(\sigma) \sqsubseteq comp_f(g_2)(\sigma)$. C.Q.F.D.

Théorème 5 *Si $f \in D_I$, alors la fonction*

$$\lambda g \in D_I.\lambda\sigma.\text{ si } f(\sigma) = \bot \text{ alors } \bot \text{ sinon } g(f(\sigma))$$

est une fonction continue de D_I dans D_I.

Demonstration. Soit g_i une suite croissante de fonctions dans D_I et soit g sa plus petite borne supérieure. Pour tout σ de *state*, $comp_f(g_i)(\sigma)$ est une suite croissante de valeurs de $state_\bot$. si $f(\sigma) = \bot$, on a $comp_f(g_i)(\sigma) = \bot$ pour tout i donc $\bigsqcup(comp_f(g_i)) = \bot$. Par ailleurs, $comp_f(g)(\sigma) = \bot$ par définition de $comp_f$, donc on a bien $comp_f(g)(\sigma) = [\bigsqcup comp_f(g_i)](\sigma)$ dans ce cas. Si $f(\sigma) \neq \bot$ alors on a la propriété suivante:

$$\begin{aligned}
[\bigsqcup comp_f(g_i)](\sigma) &= \bigsqcup comp_f(g_i)(\sigma) \\
&= \bigsqcup g_i(f(\sigma)) \\
&= g(f\sigma)
\end{aligned}$$

Le pas délicat dans cette démonstration est la première égalité: dans le membre gauche le plus petit majorant est pris dans l'ordre partiel D_I. Dans le membre droit, l'ordre partiel est pris sur $state_\perp$. Mais comme l'ordre $\sqsubseteq$ est transposé à l'ensemble des fonctions point par point, la fonction qui est le plus petit majorant est la fonction qui associe à chaque valeur le plus petit majorant. C.Q.F.D.

Théorème 6 *Si t est une fonction de $state \to bool_\perp$ et si F et G sont des fonctions croissantes et continues de D_I dans D_I alors la fonction H ci-dessous est croissante et continue:*

$$\begin{aligned} H \quad = \quad & \lambda f.\lambda\sigma. \text{ si } t(\sigma) = \perp \text{ alors } \perp \\ & \qquad \text{si } t(\sigma) = true \text{ alors } F(f)(\sigma) \text{ sinon } G(f)(\sigma). \end{aligned}$$

Démonstration. Montrons d'abord que la fonction H est croissante. Soient g_1 et g_2 deux fonctions telles que $g_1 \sqsubseteq g_2$ comparons-les point par point. Soit σ un environnement, si $t(\sigma) = \perp$, alors $H(g_1)(\sigma) = H(g_2)(\sigma) = \perp$ et on a bien $H(g_1)(\sigma) \sqsubseteq H(g_2)(\sigma)$. Si $t(\sigma) = true$, alors $H(g_1)(\sigma) = F(g_1)(\sigma)$ et $H(g_2)(\sigma) = F(g_2)(\sigma)$, mais $F(g_1) \sqsubseteq F(g_2)$ parce que F est croissante et on a donc $F(g_1)(\sigma) \sqsubseteq F(g_2)(\sigma)$ par définition de $\sqsubseteq$ sur les fonctions. Le même raisonnement s'applique si $t(\sigma) = false$, en prenant G.

Cette preuve repose sur la remarque que le test effectué dans les si-alors-sinon ne dépend pas de la valeur de la fonction f passée en argument. On effectue alors les mêmes passages aux limites dans cette preuve que pour le théorème précédent. C.Q.F.D.

Théorème 7 *Toute fonction constante est croissante et continue.*

Démonstration. C'est évident. C.Q.F.D.

On déduit des théorèmes 4, 5, 6 et 7 que pour toute fonction de test t et fonction de D_I f, la fonction Γ définie ci-dessous est continue de D_I dans D_I.

$$\begin{aligned} \Gamma \quad = \quad & \lambda g.\ \lambda\sigma. \quad \text{si } t(\sigma) = \perp \text{ alors } \perp \\ & \qquad \text{sinon} \quad \text{si} \quad t(\sigma) = true \text{ alors} \\ & \qquad\qquad\qquad\quad \text{si } f(\sigma) = \perp \text{ alors } \perp \\ & \qquad\qquad\qquad\quad \text{sinon } g(f(\sigma))) \\ & \qquad\qquad\quad \text{sinon } \sigma \end{aligned}$$

La fonction $\phi(f, t)$ utilisée à la section 1.3.2 est le plus petit point fixe de cette fonction Γ.

Exercices

8. *Démontrer que les dénotations de $assign(x, 0)$ et $sequence(while(greater(x, 0), assign(x, 0)),$ $while(greater(0, x), assign(x, 0)))$ sont les mêmes.*

9. *Démontrer que la fonction $\mathcal{I}[\![while(greater(1, 0), assign(x, x))]\!]$ est égale à la fonction $\lambda s.\perp$, qui à tout état s associe $\perp$.*

1.4 Explication intuitive

On utilise $\bot$ pour représenter une notion de valeur indéfinie et l'ordre $\sqsubseteq$ indique qu'une valeur est mieux définie qu'une autre. La notion est dégénérée pour les les valeurs atomiques (entiers, booléens), mais elle est plus significative pour les fonctions (la fonction peut être définie pour un nombre plus ou moins grand de variables). Si G est une fonction monotone de D_I dans D_I et $\bot$ est l'argument minimal de D_I, alors les fonctions $G(\bot)$, $G(G(\bot)) = G^2(\bot)$, $G^k(\bot)$ ont des ensembles de définition de plus en plus grand. Toutes ces fonctions sont des approximations du point fixe g de G et elles peuvent être utilisées pour tenter de trouver la valeur de ce point fixe en un point: si $G^k(\bot)(\sigma) \neq \bot$, alors on sait que $g(\sigma) = G^k(\bot)(\sigma)$.

L'opération d'ajouter un élément minimal dans un ensemble est habituellement effectuée par l'utilisation d'un type `option`. Dans les langages de programmation fonctionnelle la fonction ϕ s'écrit aussi très facilement. Voici par exemple comment on peut l'écrire en ocaml:

```
let rec phi t f g sigma =
  match t sigma with
    None => None
  | Some true =>
    (match f sigma with
      None => None
    | Some sigma' => g (f sigma'))
  | Some false => Some sigma
```

Dans les systèmes de démonstration sur ordinateur comme Coq, on peut définir la fonction ϕ en se reposant sur les axiomes de la logique classique, mais on obtient des fonctions qui ne s'exécute pas dans Coq. Pour faire des calculs approximatifs, on peut néanmoins remplacer toute instance de ϕ par une instance de $G^k(\bot)$ avec k suffisamment grand. Les approximations ainsi obtenues sont conservatives: si le résultat est $\bot$ (représenté par `None` habituellement), alors on sait seulement que l'exécution du programme requiert plus de k déroulement de l'une des boucles qu'il contient, et l'on ne sait pas si un résultat pourrait être obtenu avec un k plus grand. En revanche, si le résultat est une valeur (encapsulée dans `Some`) alors on sait que le programme termine et retourne cette valeur. L'encodage de la sémantique dénotationnelle en Coq en utilisant des approximations de la fonction ϕ fournit donc une technique automatique pour prouver qu'un programme s'exécute correctement, termine, et retourne une certaine valeur.

2 conclusion

Une fois que l'on a défini la fonction ϕ, on s'aperçoit que la sémantique dénotationnelle du langage est compositionnelle: la sémantique de chaque construction est obtenue par composition des sémantiques des sous-termes de cette construction.

Avoir une sémantique compositionnelle est très utile pour prouver qu'un outil de transformation de programme est correct. En effet, il suffit de démontrer que deux instructions ont la même sémantique pour que l'on ait le droit de remplacer toute instance de l'une par une instance de l'autre.