UPMC/master/info/APS-4I503
Sémantique dénotationnelle

P. Manoury

janvier 2015

La *valeur* d'un programme, et non la valeur du résultat de son évaluation, est une *fonction* qui, étant donné un état mémoire produit une valeur et une mémoire. Cette fonction est appelée *dénotation* du programme. Si Rv est le *domaine* des valeurs des valeurs en résultat et S est le domaine des mémoires (c'est-à-dire, des fonctions du domaine des adresses dans le domaine des valeurs stokées en mémoire), on définit une *fonction sémantique* $\mathbf{P} : \textsc{Prog} \to (S \to Rv \times S)$ qui calcule la dénotation du programme passé en argument. Notez que $\mathbf{P}$ est une fonction qui calcule une fonction. Définir la sémantique dénotationnelle d'un langage c'est définir la fonction $\mathbf{P}$ qui pour toute suite cs de déclarations de classes, toute suite vs de déclarations de variables et toute suite is d'instructions telles que (program cs vs is) $\in$ Prog calcule la fonction correspondant au programme (program cs vs is). On note $\mathbf{P}$[[program cs vs is]].

Comme nous en avons pris l'habitude, la fonction $\mathbf{P}$ est définie par cas de construction syntaxique des programmes. On aura ainsi des fonctions sémantiques pour les expressions, les instructions, etc. Les fonctions sémantiques sont définies par un système d'équations appellées *équations sémantiques*.

Contrairement à ce que nous avions fait pour les relations définissant la sémantique opérationnelle, les définitions des fonctions sémantiques *ne seront pas récursives*. Nous aurons à la place recours à la notion de *point fixe*.

> Comme nous l'avions fait pour la sémantique opérationnelle, nous donnerons dans un premier temps la définition de la sémantique dénotationnelle du fragment purement impératif du lanage, puis nous ajouterons les définitions de classes et l'usage de leurs instances dans un second temps.

Les notions ensemblistes utiles ainsi que celles de λ-calcul étendu nécessaires a l'écriture des fonctions sémantiques sont données en 3.

1 Fragment impératif pur

1.1 Domaines

Les *domaines* considérés en sémantique dénotationnelle sont des ensembles partiellements ordonnés, plus précisément, selon un *ordre partiel complet*. Cette notion est dûe à Dana Scott.

- N ensemble de valeurs numériques
- $B = \{\mathsf{tt}; \mathsf{ff}\}$ ensemble des valeurs booléennes
- F ensemble de fonctions de bases : $N \times N \to N$, $N \times N \to B$, $B \times B \to B$, $B \to B$, etc.
- $U = \{\mathsf{null}\}$ la valeur qui n'en est pas une
- A ensemble d'adresses
- $Dv = N \oplus B \oplus A \oplus F$ valeurs en environnement
- $E = \mathsf{id} \to Dv$ environnements
- $O = (U \cup\ B \cup N)*$ suite de valeurs en sortie
- $Sv = U \oplus B \oplus N \oplus O$ valeurs en mémoire
- $Rv = Sv \oplus \emptyset$ valeur de ≪retour≫
- $S = A \to Sv$ la mémoire (≪environnement≫ dynamique)

1.2 Expressions

La dénotation d'une expression est une fonction qui, étant donné un environnement et une mémoire calcule une valeur et une mémoire [1]. On étend cette fonction aux suites d'expressions. On définit donc les fonctions sémantiques

- $\mathbf{E} : \textsc{Expr} \to (E \to S \to (Sv \times S))$
- $\mathbf{Es} : \textsc{Expr}* \to (E \to S \to (Sv * \times S))$

Fonctions sémantiques auxiliaires On se donne une fonction de transcription des constantes lexicale vers les vaeurs booléennes et numériques (resp. β et ν) ainsi qu'une fonction de test d'égalité entre identificateurs.

- $\beta : \{\texttt{true}, \texttt{false}\} \to B$
- $\nu : \{\texttt{0}, \ldots, \texttt{9}\}* \to N$
- $\iota : \mathsf{id} \to \mathsf{id} \to B$
  $\iota = \lambda c.\lambda c'.(\mathsf{if}\ c = c'\ inB\mathsf{tt}\ inB(\mathsf{ff}))$

La dénotation des fonctions prédéfines est supposée présente dans l'environnement. On les définit ainsi :

- $\mathsf{not} = \lambda v.$ let $inB(x) = v$ in $inB(\neg x)$.
- $\mathsf{and} = \lambda v_1.\lambda v_2.$ let $inB(x), inB(y) = (v_1, v_2)$ in $inB(x \wedge y)$.
- $\mathsf{or} = \lambda v_1.\lambda v_2.$ let $inB(x), inB(y) = (v_1, v_2)$ in $inB(x \vee y)$.
- $\mathsf{add} = \lambda v_1.\lambda v_2.$ let $inN(x), inN(y) = (v_1, v_2)$ in $inN(x + y)$.
- $\mathsf{sub} = \lambda v_1.\lambda v_2.$ let $inN(x), inN(y) = (v_1, v_2)$ in $inN(x - y)$.
- $\mathsf{mul} = \lambda v_1.\lambda v_2.$ let $inN(x), inN(y) = (v_1, v_2)$ in $inN(x \times y)$.
- $\mathsf{div} = \lambda v_1.\lambda v_2.$ let $inN(x), inN(y) = (v_1, v_2)$ in $inN(x/y)$.

Constantes et fonctions prédéfinies On utilise ici les fonctions sémantiques auxiliaires définies ci-dessus.

$$\begin{array}{lcl}
\mathbf{E}[[b]]\rho\sigma & = & (\beta(b), \sigma) \\
\\
\mathbf{E}[[n]]\rho\sigma & = & (\nu(n), \sigma) \\
\mathbf{E}[[\mathsf{op}\ es]]\rho\sigma & = & \text{let } inF(\varphi) = \rho(\mathsf{op}) \text{ in} \\
 & & \text{let } vs, \sigma' = \mathbf{Es}[[es]]\rho\sigma \text{ in} \\
 & & \quad (\varphi\ vs, \sigma')
\end{array}$$

Identificateurs

$$\begin{array}{lcl}
\mathbf{E}[[x]]\rho\sigma & = & \text{let } inA(a) = \rho(x) \text{ in} \\
 & & \quad (\sigma(a), \sigma)
\end{array}$$

Suites d'expressions La dénotation d'une suite d'expressions est une fonction qui, étant donnée une suite d'expressions donne une suite de valeurs et une mémoire. Elle utilise la fonction sémantique de chacune des expressions présentes dans la suite. Par exemple, avec la suite de deux expressions $[e_1; e_2]$, on aura

$$\begin{array}{lcl}
\mathbf{Es}[[e_1; e_2]]\rho\sigma & = & \text{let } v_1, \sigma_1 = \mathbf{E}[[e_1]]\rho\sigma \text{ in} \\
 & & \text{let } v_2, \sigma_2 = \mathbf{E}[[e_2]]\rho\sigma_1 \text{ in} \\
 & & \quad ([v_1; v_2], \sigma_2)
\end{array}$$

Considérons à présent une suite plus longue, que l'on note $[e_1; e_2; \ldots]$. Supposons que $\mathbf{E}[[e_1]]\rho\sigma = (v_1, \sigma_1)$ et que $\mathbf{Es}[[[e_2; \ldots]]]\rho\sigma_1 = (vs, \sigma_n)$. La valeur de $\mathbf{Es}[[[e_1, e_2; \ldots]]$ est égale à $([v_1; vs], \sigma_n)$. Notons que l'on peut poser que la valeur de la suite vide est égale à $([], \sigma)$. La dénotaion des suites d'expressions satisfait donc l'équation *récursive* suivante :

1. Pour le fragment impératif, ma mémoire calculée par une expression est identique à celle donnée en argument. Nous la plaçons ici en prévision des appels de méthodes à venir.

$$\begin{array}{lcl}\mathbf{Es}[[es]] & = & \text{if } (es = [])\\ & & \quad ([], \sigma)\\ & & \quad (\text{let } (v, \sigma_1) = \mathbf{E}[[fst(es)]]\rho\sigma \text{ in let } (vs, \sigma') = \mathbf{Es}[[snd(es)]]\rho\sigma_1 \text{ in } ([v; vs], \sigma'))\end{array}$$

où $fst(es)$ est la première expression de es et $snd(es)$ la suite privée de ce premier élément (avec es non vide).

La dénotation des suites d'expression est donc une *fonction récursive*. Mais, en sémantique dénotationnelle, de telles équations récursives sont prohibées. Pour *dénoter des fonctions récursives* on utilise le *combinateur de point fixe*. Sa règle d'évaluation est : $!x.t = t[!x.t/x]$. Et elle permet d'encoder les fonctions récursives. Par exemple, l'expression fonctionnelle

$$!f.\lambda n.(\text{if } (n = 0)\ 1\ (n \times (f\ (n - 1))))$$

dénote la fonction factorielle. Appelons F cette expression, on a :

$$\begin{array}{lcll}(F\ 2) & = & ((!f.\lambda n.(\text{if } (n = 0)\ 1\ (n \times (f\ (n - 1))))))\ 2) & \text{définition de } F\\ & = & ((\lambda n.(\text{if } (n = 0)\ 1\ (n \times (F\ (n - 1))))))\ 2) & \text{réduction de } !f.\\ & = & (\text{if } (2 = 0)\ 1\ (2 \times (F\ (2 - 1)))) & \text{réduction de } \lambda n.\\ & = & 2 \times (F\ 1) & \text{réduction de if}\\ & = & 2 \times ((!f.\lambda n.(\text{if } (n = 0)\ 1\ (n \times (f\ (n - 1))))))\ 1) & \text{on recommence}\\ & = & 2 \times (\text{if } (1 = 0)\ 1\ (1 \times (F\ (1 - 1))))\\ & = & 2 \times 1 \times (F\ 0)\\ & = & 2 \times 1 \times (\text{if } (0 = 0)\ 1\ (0 \times (F\ (0 - 1))))\\ & = & 2 \times 1 \times 1\times\\ & = & 2\end{array}$$

On définit $\mathbf{Es}[[es]]\rho\sigma$ comme une fonction récursive sur les suites d'expressions qui a églement comme paramètre une mémoire. Cette fonction est appliquée à la suite es et la mémoire σ :

$$\begin{array}{lcl}\mathbf{Es}[[es]]\rho\sigma & = & (!f.\lambda es.\lambda\sigma.\\ & & \quad \text{if } (es = [])\\ & & \quad\quad ([], \sigma)\\ & & \quad\quad (\text{let } (v_1, \sigma_1) = \mathbf{E}[[fst(es)]]\rho\sigma \text{ in}\\ & & \quad\quad\quad \text{let } (vs, \sigma') = (f\ (snd(es))\ \sigma_1) \text{ in}\\ & & \quad\quad\quad\quad ([v_1; vs], \sigma'))\\ & & \quad es\ \sigma)\end{array}$$

1.3 Instructions

Le type de la fonction sémantique des instructions diffère un peu de celui de expressions. Si elle produit églement une mémoire, elle produit en revanche une valeur de «retour» qui permettra de discriminer les valeurs produites par l'instruction `return` dans le traitement des suites d'instructions ;

- $\mathbf{I} : \textsc{Instr} \to E \to S \to Rv \times S$
- $\mathbf{Is} : \textsc{Instr}* \to E \to S \to Rv \times S$

Affectation

$$\begin{array}{lcl}\mathbf{I}[[\texttt{setvar } x\ e]]\rho\sigma & = & \text{let } v, \sigma' = \mathbf{E}[[e]]\rho\sigma \text{ in}\\ & & \text{let } inA(a) = \rho(x) \text{ in}\\ & & \quad (in\emptyset, [\sigma'; a \mapsto v])\end{array}$$

Écriture On suppose que le flux de sortie est en mémoire et que la constante `out`, présente dans l'environnement, en donne l'adresse.

$$\begin{array}{lcl}\mathbf{I}[[\texttt{write}\ e]]\rho\sigma & = & \text{let } v, \sigma' = \mathbf{E}[[e]]\rho\sigma \text{ in} \\ & & \text{let } inO(vs) = \sigma'(\texttt{out}) \text{ in} \\ & & \quad (in\emptyset, [\sigma'; \texttt{out} \mapsto inO[vs; v]])\end{array}$$

«**Retour**» L'instruction `return` produit la valeur de son argument (qui peut être `null`) qu'il faut distinguer de la valeur `null` produite, par exemple, par l'affectation.

$$\begin{array}{lcl}\mathbf{I}[[\texttt{return}\ e]]\rho\sigma & = & \text{let } (v, \sigma') = \mathbf{E}[[e]]\rho\sigma \text{ in} \\ & & \quad (inSv(v), \sigma')\end{array}$$

Conditionnelle On utilise la construction 'case' de notre λ calcul étendu pour, à la fois, discriminer les valeurs de la condition :

$$\begin{array}{lcl}\mathbf{I}[[\texttt{if}\ e\ is_1\ is_2]] & = & \text{case } \mathbf{E}[[e]]\rho\sigma \text{ of} \\ & & \quad inB(\mathsf{tt}), \sigma' : \mathbf{Is}[[is_1]]\rho\sigma' \\ & & \mid \quad inB(\mathsf{ff}), \sigma' : \mathbf{Is}[[is_2]]\rho\sigma'\end{array}$$

Boucle La boucle est un deuxième cas d'utilisation du combinateur de point fixe pour définir une fonction sémantique. En effet, le processus d'évaluation d'une boucle est un processus récursif : il répète l'évaluation du corps de la boucle. La dénotation d'une boucle est donc une fonction qui satisfait l'équation récursive suivante :

$$\begin{array}{lcl}\mathbf{I}[[\texttt{while}\ e\ is]]\rho\sigma & = & \text{case } \mathbf{E}[[e]]\rho\sigma \text{ of} \\ & & \quad inB(\mathsf{tt}), \sigma' : \\ & & \quad\quad \text{case } \mathbf{I}[[is]]\rho\sigma' \text{ of} \\ & & \quad\quad\quad in\emptyset, \sigma'' : \mathbf{I}[[\texttt{while}\ e\ is]]\rho\sigma'' \\ & & \quad\quad \mid \quad inSv(v), \sigma'' : (inSv(v), \sigma'') \quad (1) \\ & & \mid \quad inB(\mathsf{ff}), \sigma' : (in\emptyset, \sigma') \quad (2)\end{array}$$

(1) cas de sortie prématurée de boucle par un `return`
(2) cas de sortie régulière de boucle

La dénotation de la boucle est une fonction récursive sur la mémoire :

$$\begin{array}{lcl}\mathbf{I}[[\texttt{while}\ e\ is]]\rho\sigma & = & (!w.\lambda\sigma \\ & & \quad \text{case } \mathbf{E}[[e]]\rho\sigma \text{ of} \\ & & \quad\quad inB(\mathsf{tt}), \sigma' : \\ & & \quad\quad\quad \text{case } \mathbf{I}[[is]]\rho\sigma' \text{ of} \\ & & \quad\quad\quad\quad in\emptyset, \sigma'' : (w\ \sigma'') \\ & & \quad\quad\quad \mid \quad inSv(v), \sigma'' : (v, \sigma'') \quad (1) \\ & & \quad\quad \mid \quad inB(\mathsf{ff}), \sigma' : (in\emptyset, \sigma') \quad (2) \\ & & \quad \sigma)\end{array}$$

(1) cas de sortie prématurée de boucle par un `return`
(2) cas de sortie régulière de boucle

Suites d'instructions La dénotation des suites d'instructions est, à l'instar des suites d'expressions, une fonction récursive. Elle est définie par point fixe. Elle doit stafisfaire l'équations récursive suivante :

$$\begin{array}{lcl}\mathbf{Is}[[is]]\rho, \sigma & = & \text{if } (is = []) \\ & & \quad (in\emptyset, \sigma) \\ & & \quad (\text{case } \mathbf{I}[[fst(is)]]\rho\sigma \text{ of} \\ & & \quad\quad (in\emptyset, \sigma') : \mathbf{Is}[[snd(is)]]\rho\sigma' \\ & & \quad \mid \quad (inSv(v), \sigma') : (inSv(v), \sigma'))\end{array}$$

La dénotation d'une suite d'instructions est une fonction récursive sur les listes d'instructions :

$$
\begin{array}{lcl}
\mathbf{Is}[[is]]\rho\sigma & = & (!f.\lambda is.\lambda\sigma \\
& & \quad \text{if } (is = []) \\
& & \quad\quad (in\emptyset, \sigma) \\
& & \quad\quad (\text{case } \mathbf{I}[[fst(is)]]\rho\sigma \text{ of} \\
& & \quad\quad\quad (in\emptyset, \sigma') : (f\ \sigma') \\
& & \quad\quad\quad \mid\quad (inSv(v), \sigma') : (inSv(v), \sigma')) \\
& & \quad is\ \sigma)
\end{array}
$$

1.4 Déclarations et programmes

Les déclarations de variables allouent de la mémoire et enrichissent l'environnement. La fonction sémantique attachée est récursive sur les suites de déclarations.

- $\mathbf{D} : \textsc{VarDec} \to E \to S \to (E \times S)$
- $\mathbf{Ds} : \textsc{VarDec}* \to E \to S \to (E \times S)$

$$
\begin{array}{lcl}
\mathbf{D}[[c\ x]]\rho\sigma & = & \text{let } a = alloc(,\sigma) \text{ in} \\
& & \quad ([x \mapsto inA(a); \rho], [a \mapsto \mathsf{null}; \sigma])
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{Ds}[[ds]]\rho\sigma & = & (!f.\lambda ds.\lambda(\rho,\sigma). \\
& & \quad \text{if } (ds = []) \\
& & \quad\quad (\rho, \sigma) \\
& & \quad\quad (f\ snd(ds)\ (\mathbf{D}[[fst(ds)]]\rho\sigma))) \\
& & \quad ds\ \rho\ \sigma)
\end{array}
$$

Enfin, la dénotation d'un programme s'écrit

- $\mathbf{P} : \textsc{Prog} \to (Rv \times S)$

$$
\begin{array}{lcl}
\mathbf{P}[[\texttt{program}\ ds\ is]] & = & \text{let } \rho, \sigma = \mathbf{Ds}[[ds]][][] \text{ in} \\
& & \quad \mathbf{Is}[[is]]\rho\sigma
\end{array}
$$

2 Classes et objets

Il faut à présent interpréter les déclarations de classes, la création de leurs instances et les appels aux méthodes d'instances.

Une déclaration de classe est interprétée par la fonction de création de ses instances. Une instance donne accès à l'ensemble des méthodes de la classe, y compris les méthodes héritées. Une instance est donc essentiellement un *dictionnaire* de méthodes, c'est-à-dire, un environnement. La fonction de création d'instance construit donc un environnement.

Pour leur part, lors de leurs appels, les méthodes sont évaluées dans un environnement fournissant les variables d'instances de la classe (y compris les variables héritées). Celui-ci est engendré à la création de l'instance. À l'appel de chaque méthode, cet environnment est enrichi des valeurs des paramètres d'appels et des variables locales de chacune des méthodes.

Un élément important de l'environnement d'évaluation des méthodes est l'auto-référence à l'instance sur laquelle la méthode est invoquée. Une instance est donc une *valeur récursive*. La fonction de création d'instance a pour paramètre une instance. L'auto-référence sera déterminée par un point fixe de la fonction de création d'instance (primitive `new`).

Lors de la création d'une instance, les variables d'instance sont allouées. La fonction de création d'instance altère la mémoire. Elle aura donc une mémoire en paramètre et une mémoire en résultat, en sus du dictionnaire de méthodes.

Per se, une méthode est simplement une fonction qui, étant donné l'ensemble des valeurs des paramètres d'appels et un état mémoire, calcule une valeur de «retour» et un nouvel état mémoire.

2.1 Extension des domaines

- $M = Sv^* \to S \to Sv \times S$ méthodes ;
- $I = (E \times S) \to (E \times S)$ instances ;
- $C = S \to I \times S$ classes (constructeurs d'instances).

Le résultat des déclarations de classe sont rangés dans l'environnement et les instance le sont en mémoires. On a donc

- $Dv = A \oplus F \oplus C$ valeurs en environnement ;
- $Sv = U \oplus B \oplus N \oplus I \oplus O$ valeurs en mémoire.

Contrairement à notre habitude, nous présentons la sémantique des traits objets du plus gros (déclarations de classes) au plus petit (expressions impliquants les traits objets). En effet, les déclarations effectuent un gros travail préparatoire à l'usage des instances et celles-ci se comprennent mal sans connaître celles-là.

2.2 Déclarations de classes et méthodes

Les déclarations de classes et les déclarations de méthodes produisent des environnements. La signature des fonctions sémantiques associées respectivement, aux déclarations de classes, aux suites de déclaration de classes, aux déclarations de méthodes et aux suites de déclarations de méthodes sont :

- **C** : Class $\to E \to E$
- **Cs** : Class$* \to E \to E$
- **M** : Method $\to E \to E$
- **Ms** : Method$* \to E \to E$

Déclarations de classes nous donnons les équations sémantiques des classes ≪racines≫ (qui n'étendent aucune classe) et des classes possédant une classe mère.

Nous nous donnons un identificateur réservé (`iof`) comme abréviation de `instanceof`. On lui associe la fonction sémantique auxiliare ι. L'environnement qui sera construit lors de la création d'instance donnera une valeur à `this` et `super` (lorsqu'il y aura lieu).

L'équation sémantique de **C** sans sous-classage s'écrit :

$$
\begin{array}{lll}
\mathbf{C}[[\texttt{class}\ c\ xs\ ms]]\rho & = & \text{let } \delta = \\
 & & \quad \lambda\sigma. \\
 & & \quad\quad \text{let } as = alloc(xs, \sigma) \text{ in} \\
 & & \quad\quad \text{let } \sigma' = [\sigma; as \mapsto inU(\mathsf{null})] \\
 & & \quad\quad \text{let } \gamma = \\
 & & \quad\quad\quad \lambda s. \\
 & & \quad\quad\quad\quad \text{let } \rho' = [\rho; xs \mapsto inA(as); \texttt{this} \mapsto inI(s); \texttt{iof} \mapsto inF(\iota\ c)] \text{ in} \\
 & & \quad\quad\quad\quad\quad \mathbf{Ms}[[ms]]\rho' \\
 & & \quad\quad \text{in} \\
 & & \quad\quad\quad (\gamma, \sigma') \\
 & & \text{in} \\
 & & \quad [\rho; c \mapsto inC(\delta)]
\end{array}
$$

La valeur de la classe c est la fonction δ qui prend comme paramètre une mémoire σ. Elle calcule un couple formé de la fonction γ et de la mémoire σ' qui est l'extension de σ où les variables d'instances ont été initialisées. Elle étendra la mémoire en l'état de la création de l'instance (liaison *dynamique*).

La fonction γ construit l'environnement constituant les instances de la classe. Outre les variables d'instances, la référence de l'instance elle-même et la valeur de la primitive `instanceof`, cet environnement fournira l'ensemble des méthodes ; il étendra l'environnement de déclaration de la classe (liaison *statique*).

Équation sémantique de **C** avec sous-classage :

$$
\begin{array}{lcl}
\mathbf{C}[[\texttt{class}\ c\ c'\ xs\ ms]]\rho & = & \text{let } \delta = \\
 & & \quad \lambda\sigma. \\
 & & \quad\quad \text{let } inC(\delta) = \rho(c') \text{ in} \\
 & & \quad\quad \text{let } \gamma', \sigma' = \delta(\sigma) \text{ in} \\
 & & \quad\quad \text{let } as = alloc(xs, \sigma') \text{ in} \\
 & & \quad\quad \text{let } \sigma'' = [\sigma; as \mapsto inU(\mathsf{null})] \text{ in} \\
 & & \quad\quad \text{let } \gamma = \\
 & & \quad\quad\quad \lambda s. \\
 & & \quad\quad\quad\quad \text{let } \rho' = [\rho; (\gamma'\ s); xs \mapsto inA(as); \texttt{this} \mapsto inI(s); \\
 & & \quad\quad\quad\quad\quad\quad \texttt{super} \mapsto inI(\gamma'\ s); \texttt{iof} \mapsto inF(\iota\ c)] \text{ in} \\
 & & \quad\quad\quad\quad\quad \mathbf{Ms}[[ms]]\rho' \\
 & & \quad\quad \text{in} \\
 & & \quad\quad\quad (\gamma, \sigma'') \\
 & & \text{in} \\
 & & \quad [\rho; c \mapsto inC(\delta)]
\end{array}
$$

La fonction δ' de la classe mère donne la fonction de création d'instance et alloue les variables d'instance de la classe mère (γ', σ'). Les variables d'instance de la classe son allouées dans la mémoire étendue par la création de la classe mère (σ''). La fonction γ ajoute à l'environnement celui donné par la fonction γ' de création d'instance de la classe mère appliquée à l'instance en construction (s). Elle ajoute également la valeur de `super`.

Fonction **Cs** pour les suites de déclarations de classe :

$$
\begin{array}{lcl}
\mathbf{Cs}[[\ []\ ]]\rho & = & \rho \\
\mathbf{Cs}[[cl; cls]]\rho & = & \mathbf{Cs}[[cls]](\mathbf{C}[[cl]]\rho)
\end{array}
$$

Simple itération de **Cs**

Déclarations de méthodes les déclarations de méthode étendent l'environnement des déclarations de classes avec les *fermetures* associant les corps des méthodes (suites d'instruction) aux paramètres formels et variables locales des méthodes.

$$
\begin{array}{lcl}
\mathbf{M}[[\texttt{method}\ m\ xs\ xs'\ is]]\rho & = & \text{let } \psi = \\
 & & \quad \lambda vs.\lambda\sigma. \\
 & & \quad\quad \text{let } as = alloc(xs, \sigma) \text{ in} \\
 & & \quad\quad \text{let } as' = alloc(xs', [\sigma; as \mapsto inU(\mathsf{null})]) \text{ in} \\
 & & \quad\quad\quad \mathbf{Is}[[is]][\rho; xs \mapsto inA(as); xs' \mapsto inA(as')] \\
 & & \quad\quad\quad\quad [\sigma; as \mapsto vs; as' \mapsto inU(\mathsf{null})] \\
 & & \text{in} \\
 & & \quad [\rho; m \mapsto inM(\psi)]
\end{array}
$$

La fonction **M** est simplement itérée pour les suites de déclarations de méthodes.

$$
\begin{array}{lcl}
\mathbf{Ms}[[\ []\ ]]\rho & = & \rho \\
\mathbf{Ms}[[m; ms]]\rho & = & \mathbf{Ms}[[ms]](\mathbf{M}[[m]]\rho)
\end{array}
$$

2.3 Expressions

Création d'instance la fonction de création d'instance associée à la classe est appliquée à la mémoire courrante pour obtenir un environnement fournissant les variables d'instances et le dictionnaire de méthodes ainsi que l'état mémoire associé.

$$
\begin{array}{lcl}
\mathbf{E}[[\texttt{new}\ c]]\rho\sigma & = & \text{let } inC(\delta) = \rho(c) \text{ in} \\
 & & \text{let } \gamma, \sigma' = \delta(\sigma) \text{ in} \\
 & & \quad (inI(\mathsf{fix}\ \ \gamma), \sigma')
\end{array}
$$

L'auto-référence `this` est obtenue ici par point-fixe de la fonction de création d'instance γ.

Test d'instance L'instance d'une classe (qui est un environnement) connaît la valeur de la primitive (abréviation `iof`).

$$\begin{aligned}\mathbf{E}[[\texttt{instanceof } c\ e]]\rho\sigma \quad = \quad & \text{let } inI(\gamma), \sigma' = \mathbf{E}[[e]]\rho\sigma \text{ in} \\ & (\gamma(\texttt{iof})(c), \sigma')\end{aligned}$$

Accès à un champ accès à la valeur de l'identificateur du champ dans l'instance.

$$\begin{aligned}\mathbf{E}[[\texttt{getfield } e\ x]]\rho\sigma \quad = \quad & \text{let } inI(\gamma), \sigma' = \mathbf{E}[[e]]\rho\sigma \text{ in} \\ & \text{let } inA(a) = \gamma(x) \text{ in} \\ & (\sigma'(a), \sigma')\end{aligned}$$

Appel de méthode La dénotation de la méthode (*fermeture*) est présente dans la dénotation de l'instance e.

$$\begin{aligned}\mathbf{E}[[\texttt{call } e\ m\ es]]\rho\sigma \quad = \quad & \text{let } vs, \sigma' = \mathbf{E}[[es]]\rho\sigma \text{ in} \\ & \text{let } inI(\gamma), \sigma'' = \mathbf{E}[[e]]\rho\sigma' \text{ in} \\ & \text{let } inM(\psi) = \gamma(m) \text{ in} \\ & (\psi\ vs\ \sigma'')\end{aligned}$$

Les dénotations de méthodes sont créées avec un environnement rassemblant les références à `this` et `super`. Si donc e est le mot clé `super`, cette occurrence ne doit avoir lieu que dans le corps d'une méthode. La dénotation de `super` doit être en conséquence présente dans l'environnement d'évaluation de l'appel ρ.

2.4 Instruction

L'appel de méthode a déjà été défini comme une expression. Son usage comme instruction change peu : on force sa valeur de retour à $in\emptyset$ (pour ne pas confondre un appel de méthode avec l'évaluation d'un `return`. L'autre instruction impliquant les traits objets est l'affectation d'un champ d'instance.

$$\begin{aligned}\mathbf{I}[[\texttt{call } e\ m\ es]]\rho\sigma \quad = \quad & \text{let } vs, \sigma' = \mathbf{E}[[es]]\rho\sigma \text{ in} \\ & \text{let } inI(\gamma), \sigma'' = \mathbf{E}[[e]]\rho\sigma' \text{ in} \\ & \text{let } inM(\psi) = \gamma(m) \text{ in} \\ & \text{let } \_, \sigma''' = (\psi\ vs\ \sigma'') \\ & (in\emptyset, \sigma''')\end{aligned}$$

$$\begin{aligned}\mathbf{I}[[\texttt{setfield } e_1\ x\ e_2]]\rho\sigma \quad = \quad & \text{let } v, \sigma' = \mathbf{E}[[e_2]]\rho\sigma \text{ in} \\ & \text{let } inI(\gamma), \sigma'' = \mathbf{E}[[e_1]]\rho\sigma' \text{ in} \\ & \text{let } inA(a) = \gamma(x) \text{ in} \\ & (in\emptyset, [\sigma''; a \mapsto v])\end{aligned}$$

3 Outils

3.1 Ensembles : théorie algébrique

- $x \in X$ appartenance ;
- $x \notin X$ négation de l'appartenance ;
- $\emptyset$ ensemble vide : pour tout x, $x \notin \emptyset$;
- (x, y) paire ordonnée des éléments x et y ;
- $X \times Y$ produit cartésien : $z \in X \times Y$ ssi il existe $x \in X$ et $y \in Y$ tel que $z = (x, y)$. On a que pour tout $x \in X$ et $y \in Y$, $(x, y) \in X \times Y$. On se donne $\mathsf{fst} : X \times Y \to X$ et $\mathsf{snd} : X \times Y \to Y$ tels que $\mathsf{fst}(x, y) = x$ et $\mathsf{snd}(x, y) = y$;
- X^n, avec $n \in I\!N$, produit généralisé : $X^0 = \emptyset$ et $X^{n+1} = X \times X^n$;
- $X \cup Y$ union : $z \in (X \cup Y)$ ssi $z \in X$ ou $z \in Y$;
- $\bigcup_{i \in I} E(i)$, avec $E(i)$ expression ensembliste, union généralisée, famille indicée : $z \in \bigcup_{i \in I} E(i)$ ssi il existe $i \in I$ tel que $x \in E(i)$;
- X^* suites finies d'éléments de X : $X^* = \bigcup_{i \in I\!N} X^i$.

- $X \oplus Y$ union disjointe. On se donne $inX : X \to X \oplus Y$ et $inY : Y \to X \oplus Y : z \in (X \oplus Y)$ ssi il existe $x \in X$ tel que $z = inX(x)$ ou il existe $y \in Y$ tel que $z = inY(y)$ (*injections canoniques*). On se donne $outX : X \oplus Y \to X$ et $outY : X \oplus Y \to Y$ tels que $outX(inX(x)) = x$ et $outY(inY(y)) = y$. On se donne isX et isY tels que $isX(inX(x)) = \mathsf{tt}$, $isX(inY(y)) = \mathsf{ff}$, $isY(inY(y)) = \mathsf{tt}$, $isY(inX(x)) = \mathsf{ff}$. On peut définir $inX(x) = (0, x)$ avec $x \in X$ et $inY(y) = (1, y)$ avec $y \in Y$. On pose alors $X \oplus Y = (\{0\} \times X) \cup (\{1\} \times Y)$; on a $outX(0, x) = x$ et $outY(1, y) = y$ ainsi que $isX(0, x) = \mathsf{tt}$, $isX(1, y) = \mathsf{f}$, $isY(0, x) = \mathsf{ff}$, $isY(1, y) = \mathsf{tt}$, .

3.2 Un langage de fonctions

- Base λ-calcul : $t ::= x | (t\ t) | \lambda x.t$
- *Redex* : $(\lambda x.t\ u)$ et *β-réduction* $t[u/x]$
- Macro pour les *redex* : let $x = u$ in $t \equiv (\lambda x.t\ u)$
- Constantes et opérations externes : booléens tt, ff ; entiers. Des opérations de bases aux fonction : $\lambda x.\lambda y.(x + y)$.
- Une constantes pour l'indéfini : $\bot$.
- Structure de contrôle conditionnelle : if ; avec $(\mathsf{if}\ \mathsf{tt}\ t_1\ t_2) = t_1$ et $(\mathsf{if}\ \mathsf{ff}\ t_1\ t_2) = t_2$.
- Produits et macros pour le *filtrage*. Soit $t \in X \times Y$. On pose

$$\begin{array}{rcl} \text{let } (x, y) = t \text{ in } u & \equiv & \begin{array}{l} \text{let } x = (\mathsf{fst}\ t) \text{ in} \\ \text{let } y = (\mathsf{snd}\ c) \text{ in} \\ \quad u \end{array} \end{array}$$

$$\begin{array}{rcl} \lambda(x, y).t & \equiv & \lambda c.\ \text{let } (x, y) = c \text{ in } t \end{array}$$

- Unions disjointes (types sommes) et macros pour le *filtrage*. Soit $t \in X \oplus Y$, on pose

$$\begin{array}{rcl} \text{let } inX(x) = t \text{ in } u & \equiv & \begin{array}{l} (\mathsf{if}\ isX(t) \\ \quad \text{let } x = outX(t) \text{ in } u \\ \quad \bot) \end{array} \end{array}$$

$$\begin{array}{rcl} \begin{array}{l} \text{case } t \text{ of} \\ \quad inX(x) : t_1 \\ \mid \quad inY(y) : t_2 \end{array} & \equiv & \begin{array}{l} (\mathsf{if}\ isX(t) \\ \quad \text{let } x = outX(t) \text{ in } t_1 \\ \quad (\mathsf{if}\ isY(t) \\ \quad\quad \text{let } y = outY(t) \text{ in } t_2 \\ \quad\quad \bot)) \end{array} \end{array}$$

- Produits, unions disjointes et filtrage

$$\begin{array}{rcl} \text{let } inX(x), z = t \text{ in } u & \equiv & \begin{array}{l} \text{let } inX(x) = \mathsf{fst}\ t) \text{ in} \\ \text{let } z = (\mathsf{snd}\ t) \text{ in} \\ \quad u \end{array} \end{array}$$

$$\begin{array}{rcl} \begin{array}{l} \text{case } t \text{ of} \\ \quad (inX(x), z) : t_1 \\ \quad (inY(y), z) : t_2 \end{array} & \equiv & \begin{array}{l} \text{let } (w, z) = t \text{ in} \\ \quad \text{case } w \text{ of} \\ \quad\quad inX(x) : t_1 \\ \quad\quad inY(w) : t_2 \end{array} \end{array}$$

- *Combinateur de point fixe* : fix ; tel que $(\mathsf{fix}\ \lambda x.t) = t[(\mathsf{fix}\ \lambda x.t)/x]$
  On utilisera $!x.t$ comme abréviation de l'application $(\mathsf{fix}\ \lambda x.t)$. On a alors : $!x.t = t[!x.t/x]$