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

P. Manoury

janvier 2015

L'évaluation d'un programmme produit des *valeurs* ou des *effets*. Les valeurs appartiennent aux types de base (`Int`, `Bool` ou `void`) mais sont aussi des instances de classes. Les effets sont produits par l'affectation qui modifie un *état mémoire*, la primitive d'écriture `write` qui modifie un *flux de sortie*, les déclarations modifient l'état mémoire en y introduisant de nouveaux objets. Enfin, la primitive de création d'instance `new` produit une valeur et affecte la mémoire.

Donner la sémantique opérationnelle du langage de programmation, c'est définir comment les valeurs et les effets sont obtenus par un programme en fonction des valeurs et effets obtenus par ses composants (méthodes, instructions, expressions). À l'instar des règles de typage, on donne une définition récursive des *règles d'évaluation* qui forment la sémantique opérationnelle d'un langage en se basant sur la définition récursive de la synatxe (abstraite) du langage.

> On donnera dans un premier temps la sémantique d'un fragment simplifié du langage en excluant les définitions de classes et donc l'usage des instances. C'est la sémantique des programmes de la forme
>
> `program` VarDec* Instr*
>
> Les définitions de classes et l'usage de leurs instances seront réintroduites dans un second temps.

1 Simples programmes impératifs

Les valeurs Selon le degré de précision ou de réalisme que l'on veut donner à la sémantique on considère comme domaine des valeurs numériques l'ensemble des entiers naturels, des entiers relatifs ou encore un sous ensemble d'entiers, par exemple compris entre -2^{31} et $2^{31}-1$. Pour les valeurs booélennes, on considère un enseble distinct à 2 valeurs, par exemple $\{tt, ff\}$ ou deux valeurs entières distinctes (0 et 1), voire l'ensemble des valeurs numériques dans lequel on distinguera 0.

On note V l'ensemble des valeurs

La mémoire Abstraitement, c'est une relation entre un *domaine d'adresses* et des valeurs. On pose que cette relation doit être partielle et fonctionnelle. Puisque la relation est fonctionnelle, on notrera $\sigma(a)$ la valeur «contenu» par la mémoire σ à l'adresse a. On pose que $\sigma(a) = \perp$ lorsque $a \notin \mathrm{dom}(\sigma)$.

On modélise l'allocation mémoire par une fonction abstraite *alloc* telle que si $alloc(\sigma) = a$ alors $\sigma(a) = \perp$.

On note $[\sigma; a \mapsto v]$ la fonction de domaine $\mathrm{dom}(\sigma) \cup \{a\}$ telle que

$$\begin{cases} [\sigma; a \mapsto v](a) &= v & \text{et} \\ [\sigma; a \mapsto v](a') &= \sigma(a') & \text{lorsque } a' \neq a \end{cases}$$

Par extension, lorsque as est une suite $a_1 \dots a_n$ d'adresses et vs une suite $v_1 \dots v_n$ de valeurs, on note $[\sigma; as \mapsto vs]$ la fonction de domaine $\mathrm{dom}(\sigma) \cup as$ telle que

$$\begin{cases} [\sigma; as \mapsto vs](a_i) &= v_i & \text{et} \\ [\sigma; as \mapsto vs](a') &= \sigma(a') & \text{lorsque } a' \notin as \end{cases}$$

On note $[\sigma; as \mapsto v]$ la fonction de domaine $\mathrm{dom}(\sigma) \cup as$ telle que

$$\begin{cases} [\sigma; as \mapsto v](a_i) & = & v & \text{et} \\ [\sigma; as \mapsto vs](a') & = & \sigma(a') & \text{lorsque } a' \notin as \end{cases}$$

On note $[\sigma; \sigma']$ la fonction de domaine $\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma')$ telle que

$$\begin{cases} [\sigma; \sigma'](a) & = & \sigma'(a) & \text{si } a \in \mathrm{dom}(\sigma') \\ [\sigma; \sigma'](a) & = & \sigma(a) & \text{sinon} \end{cases}$$

On notre S l'ensemble des mémoires

L'environnement Un environnement est une relation fonctionnelle partielle entre de domaine des identificateurs (Id) et le domaine des adresses. Pour les extensions d'environnements, on utilise des opérations et des notations analogues à celle utilisées pour la mémoire : $[\rho; x \mapsto a]$, $[\rho; xs \mapsto as]$, $[\rho; \rho']$.

On note E l'ensemble des environnements

1.1 Expressions pures

On qualifie de «pures» les expressions qui n'ont aucun effet sur la mémoire. Sans possibilité de définir des classes, et donc des méthodes, toutes les expressions du fragment simplifié sont pures.

Le relation sémantique concernant les expressions est définie de $(E \times M \times \textsc{Exp})$ dans V. On note $\rho, \sigma \vdash e \hookrightarrow v$. On définit par induction sur e.

Constantes Ici, on *interprètera* tous les symboles de constantes par une valeur numérique. Si $n \in \mathsf{num}$ on suppose que $\nu(n)$ nous donne la valeur entière correspondante (se définit par induction sur la suite de chiffres n). On pose la règle axiome

$$\frac{}{\rho, \sigma \vdash n \hookrightarrow \nu(n)}$$

Pour les 3 autres symboles de constantes, on pose

$$\frac{}{\rho, \sigma \vdash \texttt{null} \hookrightarrow 0} \quad \frac{}{\rho, \sigma \vdash \texttt{true} \hookrightarrow 1} \quad \frac{}{\rho, \sigma \vdash \texttt{false} \hookrightarrow 0}$$

Variables la valeur d'une variable est à l'adresse mémoire que donne l'environnement :

$$\frac{}{\rho, \sigma \vdash x \hookrightarrow \sigma(\rho(x))}$$

Opérations Nous allons donner plusieurs versions de la sémantique de *l'application* d'un opérateur.

1ère version

$$\frac{\rho, \sigma \vdash e_1 \hookrightarrow v_1 \qquad \rho, \sigma \vdash e_2 \hookrightarrow v_2}{\rho, \sigma \vdash (\texttt{add}\ e_1\ e_2) \hookrightarrow v_1 + v_2} \quad \frac{\rho, \sigma \vdash e_1 \hookrightarrow v_1 \qquad \rho, \sigma \vdash e_2 \hookrightarrow v_2}{\rho, \sigma \vdash (\texttt{sub}\ e_1\ e_2) \hookrightarrow v_1 - v_2}$$

$$\frac{\rho, \sigma \vdash e_1 \hookrightarrow v_1 \qquad \rho, \sigma \vdash e_2 \hookrightarrow v_2}{\rho, \sigma \vdash (\texttt{mul}\ e_1\ e_2) \hookrightarrow v_1 \times v_2} \quad \frac{\rho, \sigma \vdash e_1 \hookrightarrow v_1 \qquad \rho, \sigma \vdash e_2 \hookrightarrow v_2}{\rho, \sigma \vdash (\texttt{div}\ e_1\ e_2) \hookrightarrow v_1 \div v_2}$$

$$\frac{\rho, \sigma \vdash e \hookrightarrow v}{\rho, \sigma \vdash (\texttt{not}\ e) \hookrightarrow (v + 1) \bmod 2}$$

$$\frac{\rho, \sigma \vdash e_1 \hookrightarrow v_1 \qquad \rho, \sigma \vdash e_2 \hookrightarrow v_2}{\rho, \sigma \vdash (\texttt{and}\ e_1\ e_2) \hookrightarrow (v_1 \times v_2)} \quad \frac{\rho, \sigma \vdash e_1 \hookrightarrow v_1 \qquad \rho, \sigma \vdash e_2 \hookrightarrow v_2}{\rho, \sigma \vdash (\texttt{or}\ e_1\ e_2) \hookrightarrow (v_1 + v_2 + (v_1 \times v_2)) \bmod 2}$$

$$\frac{\rho,\sigma \vdash e_1 \hookrightarrow v_1 \qquad \rho,\sigma \vdash e_2 \hookrightarrow v_2}{\rho,\sigma \vdash (\texttt{eq}\ e_1\ e_2) \hookrightarrow eqz(v_1 - v_2)} \qquad \frac{\rho,\sigma \vdash e_1 \hookrightarrow v_1 \qquad \rho,\sigma \vdash e_2 \hookrightarrow v_2}{\rho,\sigma \vdash (\texttt{lt}\ e_1\ e_2) \hookrightarrow ltz(v_1 - v_2)}$$

avec $eqz(0) = 1$ et $eqz(n) = 0$ si $n \neq 0$; $ltz(n) = 1$ si $n < 0$ et $ltz(n) = 0$ sinon.

2ème version On peut factoriser cet ensemble de règles en donnant une *fonction d'évaluation* ϕ des symboles d'opération :

$$\begin{array}{lcl}
\phi(\texttt{add}) & = & \lambda x \lambda y.(x + y) \\
\phi(\texttt{sub}) & = & \lambda x \lambda y.(x - y) \\
\phi(\texttt{mul}) & = & \lambda x \lambda y.(x \times y) \\
\phi(\texttt{div}) & = & \lambda x \lambda y.(x \div y) \\
\phi(\texttt{and}) & = & \lambda x \lambda y.(x \times y) \\
\phi(\texttt{or}) & = & \lambda x \lambda y.((x + y + (x \times y)) \bmod 2) \\
\phi(\texttt{not}) & = & \lambda x.((x + 1) \bmod 2) \\
\phi(\texttt{eq}) & = & \lambda x \lambda y.(eqz(x - y)) \\
\phi(\texttt{lt}) & = & \lambda x \lambda y.(ltz(x - y))
\end{array}$$

La ≪valeur≫ des opérateurs est une fonction que l'on écrit ici en utilisant le formalisme de la λ-*notation* : si e est une expression (dans le langage du domaine des valeurs), alors $\lambda x.e$ est la fonction qui, à toute valeur v de x associe la valeur de e dans laquelle on remplace x par v. Par exemple $\lambda x.((x + 1) \bmod 2)$ appliqué à 0 est égal à la valeur de $((0 + 1) \bmod 2)$, c'est-à-dire 1.

On n'a alors plus qu'une règle

$$\frac{\rho,\sigma \vdash es \hookrightarrow vs}{\rho,\sigma \vdash (\texttt{op}\ es) \hookrightarrow \phi(\texttt{op})(vs)}$$

avec la convention que si es est la suite $e_1 \ldots e_n$ et vs, la suite $v_1 \ldots v_n$ alors $\rho,\sigma \vdash es \hookrightarrow vs$ est une abréviation pour la suite $\rho,\sigma \vdash e_1 \hookrightarrow v_1$, …, $\rho,\sigma \vdash e_n \hookrightarrow v_n$. L'écriture $\phi(\texttt{op})(vs)$ désigne l'application de la fonction $\phi(\texttt{op})$ aux valeurs vs.

3ème version La deuxième version masque un mécanisme important du processus d'évaluation de l'application d'un opérateur, ou, plus généralement, d'une fonction : celui du ≪passage des paramètres≫. On peut détailler ce mécanisme en modifiant la nature l'usage des environnements.

On étend la notion d'environnement comme une relation (toujours fonctionnelle et partielle) entre l'ensemble des identificateurs et l'ensemble des adresses et des valeurs (de Id dans $A \cup V$). On aura $\rho(x) = a$, avec $a \in A$, ou $\rho(x) = v$, avec $v \in V$ ou $\rho(x) = \perp$. On suppose que l'on sait distiguer ces 3 cas. On peut réaliser le principe d'évaluation de l'application par remplacement des *paramètres formels* par les *paramètres d'appels* en mémorisant dans l'environnement les valeurs des paramètres formels.

$$\frac{\rho,\sigma \vdash e \hookrightarrow v \qquad [\rho; x \mapsto v],\sigma \vdash t \hookrightarrow v'}{\rho,\sigma \vdash (\lambda x.t)(e) \hookrightarrow v'}$$

1.2 Instructions

L'instruction d'affectation aura un effet sur la mémoire et, par effet de composition, toutes autres les instructions le peuvent aussi. Mais comme l'instruction `return` doit produire une valeur, possiblement les instructions composées et les suites d'instructions le peuvent aussi. Enfin, l'instruction `write` produit un effet sur le flux de sortie, donc donc instructions composées et les suites d'instructions le peuvent aussi.

On modélise simplement le flux de sortie par une suite de valeurs.

Étant donné un environnement, une mémoire et un flux de sortie, la relation sémantique pour les instructions doit donc produire une valeur, une mémoire et un flot de sortie.

Il faut toutefois savoir distinguer le cas où une instruction, ou une suite d'instructions ne produit pas de valeur. Pour ce, nous complétons l'ensemble des valeurs V par une ≪non-valeur≫ distinguée que ne peut produire aucune expression. On pose $V_\emptyset = V \cup \{\emptyset\}$.

La relation sémantique pour les instructions va donc de $(E \times S \times V* \times \textsc{Instr}*)$ dans $(V_\emptyset \times S \times V*)$. On note $\rho, \sigma, o \vdash is \hookrightarrow (v, \sigma', o')$.

La suite d'instructions vide ne produit pas de valeur, n'affecte ni la mémoire, ni le flux de sortie. On pose donc

$$\frac{}{\rho, \sigma, o \vdash [] \hookrightarrow (\emptyset, \sigma, o)}$$

On donne une règle d'évaluation pour chacune des instructions du fragment considéré.

L'affectation modifie la mémoire et ne produit pas de valeur

$$\frac{\rho, \sigma \vdash e \hookrightarrow v}{\rho, \sigma, o \vdash (\texttt{set}\ x\ e) \hookrightarrow (\emptyset, [\sigma; \rho(x) \mapsto v], o)}$$

L'instruction `return` produit une valeur

$$\frac{\rho, \sigma \vdash e \hookrightarrow v}{\rho, \sigma, o \vdash (\texttt{return}\ e) \hookrightarrow (v, \sigma, o)}$$

L'écriture ajoute une valeur au flux de sortie et ne produit pas de valeur

$$\frac{\rho, \sigma \vdash e \hookrightarrow v}{\rho, \sigma, o \vdash (\texttt{write}\ e) \hookrightarrow (\emptyset, \sigma, [v; o])}$$

La conditionnelle est une structure de contrôle qui n'évalue que l'une de ses branches selon la valeur de la condition. Le résultat de l'évaluation de la conditionnelle est celui de la séquence d'instruction qui a été sélectionnée par la valeur de la condition.

$$\frac{\rho, \sigma \vdash e \hookrightarrow 1 \qquad \rho, \sigma, o \vdash is_1 \hookrightarrow (v, \sigma', o')}{\rho, \sigma, o \vdash (\texttt{if}\ e\ is_1\ is_2) \hookrightarrow (v, \sigma', o')} \qquad \frac{\rho, \sigma \vdash e \hookrightarrow 0 \qquad \rho, \sigma, o \vdash is_2 \hookrightarrow (v, \sigma', o')}{\rho, \sigma, o \vdash (\texttt{if}\ e\ is_1\ is_2) \hookrightarrow (v, \sigma', o')}$$

La boucle itère une séquence d'instruction qui forment le corps de la boucle. L'itération est contrôlée par une condition. Si celle-ci n'est pas vérifiée, la boucle ne produit pas de valeur et ne modifie ni la mémoire, ni le flux de sortie

$$\frac{\rho, \sigma \vdash e \hookrightarrow 0}{\rho, \sigma, o \vdash (\texttt{while}\ e\ is) \hookrightarrow (\emptyset, \sigma, o)}$$

Toutefois, ce contrôle nominal peut être ignoré si une itération du corps de la boucle amène à l'évaluation d'un `return`. On identifie ce cas de figure lorsque l'évaluation du corps de la boucle produit une valeur v qui n'est pas $\emptyset$. Le résultat de la boucle est alors celui de la suite d'instruction (interrompue) du corps de la boucle :

$$\frac{\rho, \sigma \vdash e \hookrightarrow 1 \qquad \rho, \sigma, o \vdash is \hookrightarrow (v, \sigma', o')}{\rho, \sigma, o \vdash (\texttt{while}\ e\ is) \hookrightarrow (v, \sigma', o') \quad \text{avec } v \neq \emptyset}$$

Si une itération n'est pas interrompue par un `return`, on procède récursivement à l'évaluation de la boucle dans le conexte produit par une itération du corps de la boucle.

$$\frac{\rho, \sigma \vdash e \hookrightarrow 1 \qquad \rho, \sigma, o \vdash is \hookrightarrow (\emptyset, \sigma', o') \qquad \rho, \sigma', o' \vdash (\texttt{while}\ e\ is) \hookrightarrow (v, \sigma'', o'')}{\rho, \sigma, o \vdash (\texttt{while}\ e\ is) \hookrightarrow (v, \sigma'', o'')}$$

Une suite non vide d'instructions produit soit le résultat de sa première instruction si celle-ci produit une valeur (c'est que dans ce cas, la première instruction a provoqué l'évaluation d'un `return`) ; soit, récursivement, le résultat du reste de ses instructions dans le contexte produit par la première d'entre elles.

$$\frac{\rho, \sigma, o \vdash i \hookrightarrow (v, \sigma', o')}{\rho, \sigma, o \vdash [i; is] \hookrightarrow (v, \sigma', o')} \text{ avec } v \neq \emptyset. \qquad \frac{\rho, \sigma, o \vdash i \hookrightarrow (\emptyset, \sigma', o') \qquad \rho, \sigma', o' \vdash is \hookrightarrow (v, \sigma'', o'')}{\rho, \sigma, o \vdash [i; is] \hookrightarrow (v, \sigma'', o'')}$$

1.3 Programme

Le bloc de déclarations des variables (locales) du programme permet de constituer l'environnement d'exécution du corps du programme. Pour chaque variable déclarée, la mémoire nécessaire est allouée. La ≪valeur≫ d'un bloc de déclarations est un couple constitué d'un environnement et d'une mémoire. L'évaluation des déclarations est définit par une relation de $(E \times M \times \text{VarDec}*)$ dans $(E \times M)$.

$$\frac{}{\rho, \sigma \vdash [] \hookrightarrow (\rho, \sigma)} \qquad \frac{\rho, \sigma \vdash (c\ x) \hookrightarrow ([\rho; x \mapsto a], [\sigma; a \mapsto \texttt{null}]) \qquad [\rho; x \mapsto a], [\sigma; a \mapsto \texttt{null}] \vdash cxs \hookrightarrow (\rho', \sigma')}{\rho, \sigma \vdash [(c\ x); cxs] \hookrightarrow (\rho', \sigma')} \text{ avec } a = alloc(\sigma)$$

L'évaluation d'un programme consiste alors simplment à créer l'environnement d'évaluation pour la suite des instructions et procéder à l'évaluation de celle-ci

$$\frac{\rho, \sigma \vdash cxs \hookrightarrow (\rho', \sigma') \qquad \rho', \sigma', [] \vdash is \hookrightarrow (v, \sigma'', o)}{\rho, \sigma \vdash (\texttt{program}\ csx\ is) \hookrightarrow (v, \sigma'', o)}$$

2 Langage à objets

Avec l'introduction des définitions de classes, de leurs méthodes et l'usage de leurs instances, le paysage sémantique évolue : l'environnement doit pouvoir fournir un descriptif des classes déclarées, l'ensemble des valeurs s'enrichit des instances de classes.

Nous resterons à un niveau assez abstrait pour décrire ces nouvelles entités.

Classes et méthodes Une classe est définie par sa classe mère (`Obj`, par défaut), ses champs et ses méthodes.

Une méthode est définie par la liste de ses paramètres, la liste de ses variables locales et la suite d'instructions du corps de la méthode. Abstraitement, une méthode est un élément de $(\mathsf{Id}* \times \mathsf{Id}* \times \text{Instr}*)$. On note $\langle xs, xs', is \rangle$ de tels triplets.
Une classe contient la déclaration de plusieurs méthodes. On se donne un *dictionnaire* de méthodes sous la forme d'une fonction $\mathsf{id} \to (\mathsf{Id}* \times \mathsf{Id}* \times \text{Instr}*)$ dont le domaine est celui des noms des méthodes.

On note M l'ensemble des dictionnaires de méthodes.

Lorsqu'une classe c étend une classe c', ses instances héritent des champs et méthodes de la classe mère. La description complète d'une classe doit pouvoir donner accès aux champs et méthodes héritées. En première approche, la description d'une classe serait une liste de champs et un dictionnaire de méthodes : les siens propres et ceux hérités.

Toutefois une classe peut redéfinir des méthodes de sa classe mère et celles-ci doivent rester accessibles avec le mot clé `super`. La représentation d'une définition de classes doit tenir compte de ce trait. Pour cela, la description d'une classe doit également mentionner sa classe mère.

Un descriptif de classe est donc un élément de $(\mathsf{Id} \times \mathsf{Id}* \times M)$ que l'on note $\langle c, xs, \mu s \rangle$.

On note C l'ensemble des descriptifs de classe

Ces éléments sont ajoutés aux environnements qui deviennent des fonctions de $\mathsf{Id} \to A \oplus C$, où $\oplus$ est *l'union disjointe*[1].

Instances Lorsqu'une classe est instanciée, ses variables de classes sont allouées. On obtient ainsi quelque chose qui ressemble à ce que nous avons pour les environnements : une association entre identificateurs et adresses. On y ajoute la référence à la classe mère qui servira pour les appels explicites aux méthodes de la classe mère avec le mot clé `super`. On peut représenter une instance par un couple de $(\mathsf{Id} \times E)$[2], que l'on note $\langle c, \rho \rangle$, où c est le nom de la classe et ρ l'environnement contenant les (adresses des) variables d'instance – augmenté de la référence à `super`.

On note I l'ensemble des instances de classes.

On pose $V = [-2^{31} \dots 2^{31} - 1] \cup I$.

2.1 Expressions *vs* instructions

Un *appel de méthode* est à la fois une *expression* et une *instruction*. Cette dualité nous pousse à changer la forme des régles d'évaluations des expressions pour prendre en compte la possibilité d'*effet* d'une expression, sur la mémoire ou le flux de sortie. La relation d'évaluation des expressions a maintenant la forme de celle des instructions : une relation de $(E \times S \times V* \times \textsc{Expr})$ dans $(V_\emptyset \times S \times V*)$. On note $\rho, \sigma, o \vdash e \hookrightarrow (v, \sigma', o')$.

La primitive `instanceof` La *valeur* d'une instance contient le nom de la classe instanciée.

$$\frac{\rho, \sigma, o \vdash e \hookrightarrow (\langle c, \_ \rangle, \sigma', o')}{\rho, \sigma, o \vdash (\texttt{instanceof}\ c\ e) \hookrightarrow (\texttt{true}, \sigma', o')} \qquad \frac{\rho, \sigma, o \vdash e \hookrightarrow (\langle c', \_ \rangle, \sigma', o')}{\rho, \sigma, o \vdash (\texttt{instanceof}\ c\ e) \hookrightarrow (\texttt{false}, \sigma', o')} \text{ avec } c \neq c'$$

La primitive `new`

$$\frac{}{\rho, \sigma, o \vdash (\texttt{new}\ c) \hookrightarrow (\langle c, [\texttt{super} \mapsto \rho(c'); xs \mapsto as] \rangle, [\sigma; as \mapsto \texttt{null}], o)} \quad \text{avec} \quad \begin{array}{l} \langle c', xs, \_ \rangle = \rho(c) \\ as = alloc(xs, \sigma) \end{array}$$

Champ d'instance

$$\frac{\rho, \sigma, o \vdash e \hookrightarrow (\langle \_, \rho' \rangle, \sigma', o')}{\rho, \sigma, o \vdash (\texttt{getfield}\ e\ x) \hookrightarrow (\sigma'(\rho'(x)), \sigma', o')}$$

Appel de méthode Les méthodes sont rendues accessibles au travers des dictionnaires, plus précisément, à travers une liste de dictionnaires.

$$\frac{\rho, \sigma, o \vdash es \hookrightarrow (vs, \sigma_1, o_1) \qquad \rho, \sigma_1, o_1 \vdash e \hookrightarrow (\langle c, \rho' \rangle, \sigma_2, o_2) \qquad \rho'', \sigma_3, o_2 \vdash is \hookrightarrow (v, \sigma_3, o_3)}{\rho, \sigma, o \vdash (\texttt{call}\ e\ m\ es) \hookrightarrow (v, \sigma_3, o_3)}$$

$$\text{avec} \quad \begin{array}{l} \rho(c) = \langle \_, \_, \mu \rangle \\ \mu(m) = \langle xs, xs', is \rangle \\ \rho'' = [\rho; \rho'; xs \mapsto as; xs' \mapsto as'] \\ \sigma_3 = [\sigma_2; as \mapsto vs; as' \mapsto \texttt{null}] \\ as = alloc(xs, \sigma_1) \\ as' = alloc(xs', [\sigma_1; as \mapsto \texttt{null}]) \end{array}$$

1. L'union *disjointe* est un *artefact* ensembliste qui permet de reconnaître l'ensemble d'origine des éléments réunis.
2. *stricto sensu*, un environnement devient une fonction de $\mathsf{Id} \cup \{\texttt{super}\} \to V$

$$\frac{\rho, \sigma, o \vdash es \hookrightarrow (vs, \sigma_1, o_1) \qquad \rho', \sigma_2, o_2 \vdash is \hookrightarrow (v, \sigma_3, o_3)}{\begin{array}{r} \rho, \sigma, o \vdash (\texttt{call super } m\ es) \hookrightarrow (v, \sigma_3, o_3) \\ \text{avec} \quad \begin{array}{l} \rho(\texttt{super}) = \langle \_, \_, \mu \rangle \\ \mu(m) = \langle xs, xs', is \rangle \\ \rho' = [\rho; xs \mapsto as; xs' \mapsto as'] \\ \sigma_2 = [\sigma_1; as \mapsto vs; as' \mapsto \texttt{null}] \\ as = alloc(xs, \sigma_1) \\ as' = alloc(xs', [\sigma_1; as \mapsto \texttt{null}]) \end{array} \end{array}}$$

NB : appel *par valaur*, réduction par la droite.

2.2 Déclarations de classes et méthodes

Méthodes Les déclarations de méthodes construisent des dictionnaires.

$$\frac{}{\mu \vdash (\texttt{method } m\ vs\ \_\ vs'\ is) \hookrightarrow [\mu; m \mapsto \langle xs, xs', is \rangle]}$$

avec xs et xs' les identificateurs déclarés dans vs er vs'

$$\frac{}{\mu \vdash [] \hookrightarrow \mu} \qquad \frac{\mu \vdash mt \hookrightarrow \mu' \qquad \mu' \vdash mts \hookrightarrow \mu''}{\mu \vdash [mt; mts] \hookrightarrow \mu''}$$

Classes Les déclarations de classes étendent l'environnement. La description créée pour une classe embarque les champs et méthodes héritées.

$$\frac{\mu' \vdash mts \hookrightarrow \mu}{\begin{array}{r} \rho \vdash (\texttt{class } c\ c'\ xs\ mts) \hookrightarrow [\rho; c \mapsto \langle c', [xs'; xs], \mu \rangle] \\ \text{avec } \rho(c') = \langle \_, \_, \mu' \rangle \end{array}}$$

$$\frac{}{\rho \vdash [] \hookrightarrow \rho} \qquad \frac{\rho \vdash cl \hookrightarrow \rho' \qquad \rho' \vdash cls \hookrightarrow \rho''}{\rho \vdash [cl; cls] \hookrightarrow \rho''}$$

2.3 Programmes

$$\frac{\rho_0 \vdash cls \hookrightarrow \rho \qquad \rho, [] \vdash cxs \hookrightarrow (\rho', \sigma) \qquad \rho', \sigma, [] \vdash is \hookrightarrow (v, \sigma', o)}{\rho_0 \vdash (\texttt{program } cls\ cxs\ is) \hookrightarrow (v, \sigma', o)}$$

Où ρ_0 est soit l'environnement vide, soit l'environnement contenant les définitions de opérateurs prédéfinis.