MI030 — APS
Analyse des programmes et sémantique

© Jacques Malenfant, 2010–2014

avec la participation initiale d'**Olena Rogovchenko**

Université Pierre et Marie Curie
UFR 919 Ingénierie
Jacques.Malenfant@lip6.fr

Cours 3 - 4

Introduction aux sémantiques opérationnelles structurelles

Concepts centraux I

- Les sémantiques opérationnelles, dont font partie les sémantiques opérationnelles structurelles, cherchent à exprimer la signification des programmes, c'est-à-dire ce qu'ils calculent, en fonction des occurrences des différentes constructions du langage qui y apparaissent.
- Plus spécifiquement, les sémantiques opérationnelles se distinguent des autres formes de sémantiques par le fait qu'elles cherchent à capturer le *comment*, c'est-à-dire les étapes (plus ou moins) élémentaires de l'exécution du programme.

Concepts centraux II

- Il n'y a pas de définition absolue d'*étape élémentaire* ; une sémantique est définie en fonction d'une machine virtuelle qui détermine l'ensemble des opérations élémentaires qui seraient directement réalisables par une machine réelle.

Du modèle de Von Neumann à la sémantique d'un programme I

- Le modèle de calcul dominant dans les machines réelles demeure celui de Von Neumann.
- Il se caractérise par le fait que la machine a un état, et que l'exécution des instructions de la machine se concrétise par une modification de cet état.
- La plupart des machines virtuelles utilisées dans les SOS se fondent sur ce fonctionnement général, état - transition.

Du modèle de Von Neumann à la sémantique d'un programme II

- La définition de la machine virtuelle utilise une représentation des configurations ou états possibles, et une représentation des instructions comme transitions entre états.
- Le niveau général d'abstraction d'une machine virtuelle dépend du niveau de détail avec lequel les configurations et les transitions sont décrites. Toutes cependant s'éloignent des limitations physiques des machines rélles, comme la taille de la mémoire.
- Une bonne machine virtuelle réalise un compromis entre finesse de la description et complexité des sémantiques résultantes.

Du modèle de Von Neumann à la sémantique d'un programme III

  - Un trop haut niveau d'abstraction laisserait beaucoup de choses non-précisées car demeurant du niveau de la réalisation de la machine virtuelle.
  - Un niveau trop bas demanderait une sémantique très complexe pour traiter toute la finesse de la description, la rendant ainsi plus difficile à comprendre, et éventuellement trop liée à une machine virtuelle précise.
- La sémantique d'un programme capture donc une fonction prenant une configuration initiale de la machine et rend une configuration finale après exécution du programme.

Sémantique opérationnelle structurelle I

- Introduite par Plotkin au début des années ’80, la sémantique opérationnelle structurelle cherche à voir les concepts d’états et de transitions sous l’angle d’un système de déduction.
- Des règles d’inférence permettent de raisonner sur les transitions entre états. On utilise donc des règles d’inférence de la déduction naturelle comme celles utilisées au cours précédent pour définir la vérification des types.

Sémantique opérationnelle structurelle II

- Un principe de construction important consiste à définir ces transitions à partir des constructions du langage, à raison d’une règle d’inférence par type de construction (au moins), permettant de déduire la transition à réaliser pour la construction à partir des transitions opérées par ses sous-constructions ;
- Fondée sur la syntaxe abstraite du langage, ces sémantiques autorisent l’utilisation de la technique d’induction structurelle pour prouver des propriétés sur les programmes.

« *Big step* » et « *Small step* » I

- On distingue deux grandes approches dans les SOS.
- L’approche « *small step* » consiste à découper la signification des constructions par des étapes de très faible niveau d’abstraction correspondant à des instructions élémentaires très proches d’une machine réelle.

« *Big step* » et « *Small step* » II

- L’approche « *big step* » a plutôt tendance à définir la sémantique des constructions à partir des transitions globales nécessaires pour chacune de leurs sous-constructions immédiates. Elles ont donc tendance à masquer pour la plupart des constructions composées les étapes élementaires, reléguant ces dernières aux constructions de bases.
- Le choix entre les deux approches dépend de l’objectif du sémanticien.
  - Une sémantique « *small step* » lui permettra d’exprimer finement des phénomènes liés à l’ordre d’exécution des étapes élémentaires,

« *Big step* » et « *Small step* » III

- alors qu'une sémantique « *big step* » permettra plus facilement de raisonner par induction structurelle, mais au prix d'une moins grande précision dans la description opérationnelle de la signification des constructions.

Preuve par induction structurelle I

- À titre d'exemple de la technique d'induction structurelle, nous allons prouver une propriété syntaxique simple définie par le lemme suivant :

Lemme

Pour toute expression arithmétique ou logique n'utilisant pas d'opérateur unaire, le nombre d'opérandes est strictement supérieur au nombre d'opérateurs, c'est-à-dire $\#rators(e) < \#rands(e)$.

Preuve par induction structurelle II

- Notons que la grammaire abstraite donnée au cours 1 peut être reformulée pour les besoins de notre exemple sous la forme d'un système d'inférence où les règles permettent de raisonner sur l'appartenance d'un sous-arbre de syntaxe abstraite à une catégorie syntaxique donnée à partir de ses propres sous-arbres. Par exemple :

$$\frac{e_1 \in exp \qquad e_2 \in exp}{\texttt{plus}\ e_1\ e_2 \in exp}$$

Preuve par induction structurelle III

- On peut donc appliquer la technique d'induction structurelle pour prouver le lemme précedent qui ne porte que sur la syntaxe du programme.
  Preuve :
  Base : Prouver le cas de base pour les expressions demande de considérer les expressions *int*, `true`, `false`, id, `readfield`. Pour chacun de ces types d'expressions, on constate qu'il n'y a pas d'opérateurs mais qu'elles représentent une opérande. On a donc bien alors $0 = \#rators(e) < \#rands(e) >= 1$.

Preuve par induction structurelle IV

Induction : On doit considérer chacune des constructions composées sous l'hypothèse d'induction :

$$\#rators(e_1) < \#rands(e_1) \text{ et } \#rators(e_2) < \#rands(e_2)$$

Cas `instanceof` e_1 e_2 : Par l'hypothèse d'induction, on a :

$$\#rators(e_1) + 1 \leq \#rands(e_1)$$

$$\#rators(e_2) + 1 \leq \#rands(e_2)$$

Preuve par induction structurelle V

Pour $e =$ `instanceof` e_1 e_2, on a :

$$\#rators(e) = \#rators(e_1) + \#rators(e_2) + 1$$

$$\#rands(e) = \#rands(e_1) + \#rands(e_2)$$

On a donc bien

$$\begin{aligned}\#rators(e) &= \#rators(e_1) + \#rators(e_2) + 1 < \\ &\#rators(e_1) + \#rators(e_2) + 2 = \\ &(\#rators(e_1) + 1) + (\#rators(e_2) + 1) \leq \\ &\#rands(e_1) + \#rands(e_2) = \#rands(e)\end{aligned}$$

Preuve par induction structurelle VI

Cas `plus` e_1 e_2, `minus` e_1 e_2, `times` e_1 e_2, `equal` e_1 e_2, `and` e_1 e_2, `or` e_1 e_2, `less` e_1 e_2 : idem. □

- De telles preuves pourront également être faites sur la base de la sémantique des programmes pour prouver des propriétés sur leur exécution.

Sémantique des expressions arithmétiques

- Il s'agit, pour chaque type d'expression, de montrer comment produire la valeur rendue par les occurences de ce type d'expression dans le programme.
- Par exemple, l'addition prend des sous-expressions dont les valeurs doivent être obtenues, puis additionnées pour donner le résultat de l'expression d'addition.

Les expressions arithmétiques simples

- Pour définir la sémantique, il faut avoir la syntaxe abstraite du langage. Considérons comme dans le cours précédent :

$$e \ ::= \ e + e \mid e - e \mid e * e \mid e \,/\, e \mid \texttt{int} \mid \texttt{real}$$

- Le résultat de la sémantique sera la valeur donnée par l'expression.
- Pour définir ce résultat, on suppose qu'il s'agit d'une valeur de l'ensemble des entiers ou des réels, les deux étant munis des opérations arithmétiques habituelles (addition, soustraction, multiplication, division).
- Pour gérer la conversion des entiers aux réels, on suppose l'existence d'une opération explicite $\Rightarrow_{\mathbb{R}} : \mathbb{Z} \to \mathbb{R}$.

Règles d'inférences pour la sémantique I

- Les règles d'inférence vont servir à produire le résultat du programme.
- Il faut donc définir une relation, notée comme précédemment $\to$. Les règles permettent donc de raisonner (ici pour enchaîner) sur des étapes d'évaluation de la forme :

$$e \to \mathbb{Z} \oplus \mathbb{R}$$

- Elles sont définies également pour chaque construction de la grammaire abstraite.

Règles d'inférences pour la sémantique II

$$\texttt{int}\ n \to n \qquad \texttt{real}\ r \to r$$

$$\frac{e_1 \to v_1 \quad e_2 \to v_2}{e_1 + e_2 \to v_1 +_{\mathbb{Z}} v_2} \; v_1 \in \mathbb{Z} \wedge v_2 \in \mathbb{Z}$$

$$\frac{e_1 \to v_1 \quad e_2 \to v_2}{e_1 + e_2 \to \Rightarrow_{\mathbb{R}} (v_1) +_{\mathbb{R}} v_2} \; v_1 \in \mathbb{Z} \wedge v_2 \in \mathbb{R}$$

$$\frac{e_1 \to v_1 \quad e_2 \to v_2}{e_1 + e_2 \to v_1 +_{\mathbb{R}} \Rightarrow_{\mathbb{R}} (v_2)} \; v_1 \in \mathbb{R} \wedge v_2 \in \mathbb{Z}$$

$$\frac{e_1 \to v_1 \quad e_2 \to v_2}{e_1 + e_2 \to v_1 +_{\mathbb{R}} v_2} \; v_1 \in \mathbb{R} \wedge v_2 \in \mathbb{R}$$

- Les règles pour les autres expressions composées sont similaires, à l'opérateur arithmétique près.

Évaluation d'une expression en particulier

- Comme pour le typage, l'évaluation d'une expression se fait en chaînant les règles d'inférence sous la forme d'un *arbre d'inférence*.
- Pour l'expression $(5+3)*2,5$, cela donne (les parenthèses ne font pas partie de la notation, mais sont là pour écrire des arbres de syntaxe abstraite *à plat* mais de manière non-ambigüe) :

$$\frac{\dfrac{5 \to 5 \qquad 3 \to 3}{(5+3) \to 5 +_{\mathbb{Z}} 3} \qquad 2,5 \to 2,5}{(5+3)*2,5 \to \Rightarrow_{\mathbb{R}} (8) \times_{\mathbb{R}} 2,5 = 20,0}$$

Introduction des variables I

- Les expressions arithmétiques précédentes ne comportent pas de variables. Comme pour le cas du typage, que se passe-t-il si on en ajoute ?
  La grammaire abstraite :

$$e \ ::= \ e + e \mid e - e \mid e * e \mid e \,/\, e \mid \texttt{int} \mid \texttt{real} \mid \texttt{id}$$

- Pour évaluer les références aux variables, il faut savoir récupérer la valeur associé à chacune des variables

Introduction des variables II

- Récupérer la valeur d'une variable peut se faire par un environnement. Notons ρ : $Id \to \mathbb{Z} \oplus \mathbb{R}$ la fonction associant à chaque variable la valeur qui lui a été donnée.
- Les règles d'inférence vont maintenant être écrites pour une relation d'évaluation dans le contexte d'un environnement de la forme :

$$\rho \vdash e \to \mathbb{Z} \oplus \mathbb{R}$$

- Pour le cas des variables, on ajoute la règle :

Introduction des variables III

$$\rho \vdash id \to \rho(id)$$

- Nous allons voir dans le cas de BOPL comment le système d'inférence permet de créer graduellement les environnements par l'analyse des déclarations et l'évaluation des affectations dans le programme.

Expressions arithmétiques en « *small step* » I

- Le nom « *small step* » vient du fait que les règles procèdent par étapes de réductions plus limitées que dans le cas des sémantiques dites « *big step* » :

$$\frac{e_1 \rightarrow n_1}{e_1 + e_2 \rightarrow n_1 + e_2}$$

$$\frac{e_2 \rightarrow n_2}{n_1 + e_2 \rightarrow n_1 + n_2}$$

$$\frac{}{n_1 + n_2 \rightarrow \mathit{Apply}(+, n_1, n_2)}$$

Expressions arithmétiques en « *small step* » II

- Notons que n_1 et n_2 ne sont pas des nombres dans $\mathbb{Z}$ ou $\mathbb{R}$, mais bel et bien des constantes numériques de la syntaxe abstraite.
- C'est-à-dire qu'ici, la relation $\rightarrow$ est considérée comme prenant une expression et rendant une expression, équivalente mais la plus réduite possible.
- La fonction *Apply* ne représente donc pas un calcul dans $\mathbb{Z}$ ou $\mathbb{R}$, mais plutôt une réécriture dans le domaine des expressions de la syntaxe abstraite.

Principes généraux I

- Comme pour la vérification des types, la sémantique des programmes BOPL se caractérise par le traitement des déclarations des classes puis l'exécution du corps du programme, éventuellement dans le contexte de variables locales.
- Les déclarations de classes vont devoir être traitées pour donner une représentation de ces dernières accessible pour définir plus loin la sémantique des expressions comme `new`, ou encore les accès aux variables d'instance et l'appel de méthodes ; la représentation d'une classe doit permettre de connaître
  - toutes les variables détenues par une instance de cette classe pour pouvoir les créer correctement ;

Principes généraux II

- la défintion de toutes les méthodes déclarées dans la classe, pour pouvoir les exécuter lors d'un appel de méthodes ; et,
- la superclasse pour trouver les méthodes héritées dans la représentation de cette dernière ou de ses propres superclasses.

- La sémantique d'une expression `new` doit résulter dans un objet ; les objets en tant que valeurs manipulables dans le langage doivent donc aussi être représentés.
- La sémantique de l'appel de méthode sur un objet donné nécessite de pouvoir trouver le code de la méthode à appliquer, ce qui impose de gérer le lien entre les objets et leur classe d'instantiation, pour trouver la méthode à utiliser.

Principes généraux III

- L'existence d'expressions `self` et `super` impose également la bonne définition de l'objet courant (`self`) et la classe de déclaration de la méthode courante (`super`) dans le contexte d'exécution de la méthode
- L'existence d'une instruction `return` nécessite non seulement de gérer le calcul de la valeur à retourner de la méthode mais également le fait de sortir immédiatement de la méthode sans exécuter les instructions suivantes dans la méthode.
- L'existence d'une instruction `writeln` requiert de gérer l'état du flux de sortie que nous allons nommer *out*.

Valeurs manipulées I

Entiers : des valeurs dans $\mathbb{Z}$.

Booléens : des valeurs dans $T = \{\texttt{true}, \texttt{false}\}$.

Identifiants d'objets : un ensemble d'identifiants, à raison d'un par objet utilisé dans le programme dans un ensemble *Oid*.

Identifiants de classes : parmi les identifiants utilisables dans un programme, les identifiants de classes peuvent être résultats des expressions de classes (*cexp*).

- $isType : Id \times \rho \to T$ est une fonction permettant de distinguer les identifiants de classe des autres identifiants.

Valeurs manipulées II

- En résumé, les valeurs manipulables, c'est-à-dire pouvant être le résultat d'un calcul en BOPL sont :

$$V = \mathbb{Z} \oplus T \oplus Oid \oplus Id$$

muni des fonctions $def : V \to T$ et $undef : V \to T$.

Pourquoi trois éléments dans le contexte ? I

- Comme dans le cas de la vérification des types, et comme il a été illustré par le cas des variables précédemment, la définition de la sémantique des programmes va demander l'utilisation d'un contexte pour mémoriser les liaisons des variables et autres identifiants.
- La solution la plus simple simple paraît être d'avoir une simple fonction des identifiants (*Id*) vers les valeurs manipulées.
- Mais cette solution est insuffisante dans le cas des variables, car il est tout à fait possible d'utiliser le même nom de variable en plusieurs endroits du programme.

Pourquoi trois éléments dans le contexte ? II

- On introduit donc deux éléments de contexte :
  1. l'environnement, pour gérer les liaisons locales des variables vers les adresses des mots mémoire qui vont contenir leurs valeurs, et
  2. la mémoire, pour gérer les liaison entre adresses et valeurs manipulables.
- L'environnement peut donc être défini comme une fonction :
  - $\rho : \mathit{Id} \to \mathit{Address}$, c'est-à-dire que pour obtenir l'adresse liée à un identifiant id, il suffit de faire $\rho(\mathit{id})$.
- La mémoire (« *store* ») peut donc être définie comme une fonction :
  - $\sigma : \mathit{Address} \to V$, c'est-à-dire que pour obtenir la valeur associée à une adresse a, il suffit de faire $\sigma(a)$.

Pourquoi trois éléments dans le contexte ? III

- Quelques fonctions sont introduites pour compléter l'utilisation de la mémoire :
  - *allocate* : $\sigma \to \mathit{Address}$ permet d'allouer un nouvel espace mémoire et retourne l'adresse de ce dernier.
  - *createOid* : $\mathit{Address} \to \mathit{Oid}$ permet de créer un identifiant d'objet à partir de l'adresse à laquelle l'objet correspondant va être mémorisé.
  - @ : $\mathit{Oid} \to \mathit{Address}$ permet de retrouver l'adresse correspondant à un identifiant d'objets.
  - *allocateAll* : $\mathit{Id}^* \times \rho \times \sigma \to \rho \times \sigma$ alloue les variables et retourne les environnements et mémoires après allocations.
  - *storeAll* : $\mathit{Id}^* \times V^* \times \rho \times \sigma \to \rho \times \sigma$ alloue les variables et les initialise avec les valeurs passées en argument.

Pourquoi trois éléments dans le contexte ? IV

- La représentation de l'environnement et de la mémoire utilise la notion de fonction partielle, et la notation $f[x \mapsto v]$ est utilisée pour représenter l'extension d'une fonction f pour l'entrée x qui est défini comme retournant v. Si l'entrée x est déjà définie dans f, la nouvelle liaison est masquante.
- Le troisième élément de contexte est le flux de sortie *out* qui est géré comme une chaîne de caractère. L'opération de concaténation des chaînes, notée $+$, sera utilisée pour accumuler les chaînes dans le flux de sortie.

Gestion du contexte

- Contexte et sémantique des instructions : les instructions peuvent modifier la mémoire et le flux de sortie (`writeln`) mais pas l'environnement.
- Contexte et sémantique des expressions : les expressions ne devraient pas pouvoir changer quoi que ce soit au contexte. Cependant, parmi les expressions se trouve l'appel de méthode, et le corps des méthodes peut, lui, modifier l'état de la mémoire et du flux de sortie par ses instructions (`writeField` et `writeln`).

Représentation des classes I

- À l'exécution, deux opérations nécessitent une description du contenu des classes :
  1. le `new` demande à connaître les champs déclarés et hérités par la classe pour créer l'objet avec tous les champs qui lui reviennent ;
  2. l'appel de méthode demande à connaître les méthodes déclarées par les classes, le lien d'héritage permettant de naviguer des classes vers leurs superclasses pour trouver les méthodes héritées.
- La représentation d'une classe sera donc un tuple $class(id, id_s, id^*, md)$ où
  1. id est l'identifiant de la classe,
  2. id_s est l'identifiant de la superclasse,
  3. id^* est la liste des variables déclarées et héritées par la classe, et

Représentation des classes II

4. md est un dictionnaire des méthodes déclarées par la classe, dont la forme sera donnée un peu plus loin.

- Comme les identifiants de classe sont uniques dans un programme et que les classes ne sont pas modifiables durant l'exécution d'un programme, le plus simple sera d'introduire ces dernières directement dans l'environnement, et non dans la mémoire, ce qui modifie notre définition de l'environnement pour donner :

$$\rho : Id \rightarrow Classes \oplus Address$$

- Quelques fonctions d'accès sont définies pour faciliter la manipulation des classes dans les règles :
  - $lookForMethod : Id_C \times Id_m \times \rho \rightarrow Method$ retourne la méthode Id_m associée à la classe Id_C.

Représentation des classes III

- $getSuper : Id_C \times \rho \rightarrow Id_S$ retourne l'identifiant Id_S de la superclasse d'une classe Id_C.
- $inheritsFrom : Id_1 \times Id_2 \times \rho \rightarrow T$ retourne vrai si la classe Id_1 hérite directement ou indirectement de la classe Id_2, avec $Id_1 \neq Id_2$.
- $getOwnedFields : Id \times \rho \rightarrow Id^*$ donne les champs déclarés et hérités par une classe.

Représentation des objets I

- Un objet doit connaître son identifiant unique, l'identifiant de sa classe d'instantiation et les valeurs qu'il associe aux champs déclarés ou hérités par sa classe. Il est représenté par un tuple $object(oid, id, fields)$ où
  - oid est l'identifiant d'objet unique,
  - id est l'identifiant de la classe d'instantiation, et
  - $fields : id \to V$ donne les valeurs associées aux champs par l'objet.
- Quelques fonctions sont définies pour faciliter la manipulation des objets :

Représentation des objets II

  - $classOf : Oid \times \sigma \to Id$ retourne l'identifiant de la classe d'instantiation d'un objet ;
  - $allocateFields : Id^* \to Id \to V$ crée la fonction $fields$ initiale à partir des identifiants des champs ;
  - $fieldsOf : Object \to (Id \to V)$ retourne l'association des champs à leurs valeurs d'un objet.
  - $writeField : Oid \times Id \times V \times \sigma \to \sigma$ rend une nouvelle mémoire où la valeur d'in champ dans un objet a été modifiée pour prendre une nouvelle valeur.
- La mémoire va devoir contenir des objets, donc on doit modifier la définition de la mémoire pour adopter :

$$\sigma : Address \to V \oplus Object$$

Représentation des méthodes

- Pour une méthode, de manière à pouvoir l'appliquer, il faut connaître l'identifiant de sa classe de déclaration, la liste des identifiants donnés en paramètres formels, la liste des identifiants des variables locales et les instructions formant son corps. On les représente donc par un tuple $method(id, id_1^*, id_2^*, inst)$ où
  - id est l'identifiant de la classe déclarant cette méthode ;
  - id_1^* sont les identifiants des paramètres formels ;
  - id_2^* sont les identifiants des variables locales ; et,
  - $inst$ est le corps de la méthode.

Gestion du `self` et du `super` I

- Lors de l'exécution d'une méthode, on peut rencontrer les expressions `self` et `super`.
- Le self doit permettre de récupérer l'objet qui exécute la méthode, alors que le super doit permettre de retrouver la superclasse immédiate de la classe de déclaration de la méthode à partir de laquelle la recherche de méthode doit se faire.

Gestion du `self` et du `super` II

- Comme ces deux valeurs sont connues au moment du lancement de l'exécution de la méthode, et qu'elles ne changent pas pendant cette exécution, nous introduisons les liaisons de ces deux identifiants à leur valeur dans l'environnement utilisé pour exécuter le corps de la méthode.
- Pour tenir compte de l'ajout de l'identifiant d'objet de *self* et de l'identifiant de la superclasse de la classe de définition de la méthode dans l'environnement, on modifie à nouveau la définition de ρ :

$$\rho : \mathit{Id} \to V \oplus \mathit{Classes} \oplus \mathit{Oid}$$

Gestion de l'instruction `return`

- L'instruction `return` a deux effets :
  1. donner la valeur à retourner pour l'appel de la méthode, et
  2. terminer l'exécution du corps d'une méthode.
- Pour gérer le `return`, il faut donc que la sémantique des instructions puissent donner la valeur de retour quand on se trouve dans le corps d'une méthode. La valeur de retour fera donc partie des résultats des instructions dans le corps d'une méthode.

Sémantique de `program`

- Relation $\mathit{program} \to \mathit{out}$ définie par les règles :

$$\frac{\langle \mathit{class}^*, \varnothing\rangle \to \rho \qquad \langle \mathit{var}^*, \rho, \varnothing\rangle \to \langle \rho', \sigma'\rangle \qquad \rho' \vdash \langle \mathit{inst}, \sigma', \varnothing\rangle \to \langle \sigma'', \mathit{out}\rangle}{\texttt{program}\ \mathit{class}^*\ \mathit{var}^*\ \mathit{inst} \to \mathit{out}} \quad (1)$$

$$\frac{\langle \mathit{class}^*, \varnothing\rangle \to \rho \qquad \rho \vdash \langle \mathit{inst}, \varnothing, \varnothing\rangle \to \langle \sigma', \mathit{out}\rangle}{\texttt{program}\ \mathit{class}^*\ \mathit{inst} \to \mathit{out}} \quad (2)$$

Sémantique des déclarations de classes I

- La sémantique de la liste des classes est d'abord donnée par la relation $\langle \mathit{class}^*, \rho\rangle \to \rho'$ définie par la règle :

$$\frac{\langle \uparrow \mathit{class}^*, \rho\rangle \to \rho'' \qquad \langle \downarrow \mathit{class}^*, \rho''\rangle \to \rho'}{\langle \mathit{class}^*, \rho\rangle \to \rho'} \quad (3)$$

- Alors que la sémantique d'une classe est définie par la relation $\langle \mathit{class}, \rho\rangle \to \rho'$ mettant à jour l'environnement comme il se doit :

Sémantique des déclarations de classes II

$$\frac{\rho \vdash \langle cexp, \varnothing, \varnothing\rangle \rightarrow \langle id_s, \sigma', out\rangle \qquad id \vdash \langle method^*, \varnothing\rangle \rightarrow md}{\langle \texttt{class}\ id\ cexp\ var^*\ method^*, \rho\rangle \rightarrow \rho[id \mapsto class(id, id_s, extractIds(var^*) + inheritedFields(id_s, \rho), md)]} \quad (4)$$

Sémantique des déclarations de méthodes I

- Pour la sémantique de la liste des méthodes, il faut créer successivement chaque entrée dans le dictionnaire de méthodes :

$$\frac{id \vdash \langle \uparrow method^*, md\rangle \rightarrow md'' \qquad id \vdash \langle \downarrow method^*, md''\rangle \rightarrow md'}{id \vdash \langle method^*, md\rangle \rightarrow md'} \quad (5)$$

Sémantique des déclarations de méthodes II

- Pour chaque méthode individuellement, il faut créer la structure déjà décrite et qui est de même nature qu'une fermeture. La relation $id \vdash \langle method, dm\rangle \rightarrow dm'$ permet de créer cette structure :

$$id \vdash \langle \texttt{method}\ id_m\ var_f^*\ cexp\ var_L^*\ inst, dm\rangle \rightarrow dm[id_m \mapsto method(id, extractIds(var_f^*), extractIds(var_L^*), inst)] \quad (6)$$

- La fonction *extractIds* extrait les identifiants des structures syntaxiques `var` apparaissant dans le programme.

Sémantique des déclarations de variables locales I

- La relation $\langle \texttt{var}\ cexp\ id, \rho, \sigma\rangle \rightarrow \langle \rho, \sigma\rangle$ définit la sémantique des déclarations de variables du programme en faisant les allocations et liaisons nécessaires :

$$\langle \texttt{var}\ cexp\ id, \rho, \sigma\rangle \rightarrow \langle \rho[id \mapsto a], \sigma[a \mapsto \varepsilon]\rangle \qquad \text{where } a = allocate(\sigma) \quad (7)$$

Note : Les autres formes de variables (instance, paramètres formels et variables locales aux méthodes) sont traitées différemment.

Sémantique des instructions I

- Les instructions peuvent modifier la mémoire et la sortie standard. Leur sémantique est donc définie par une relation $\rho \vdash \langle inst, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle$:

$$\frac{\rho \vdash \langle inst_1, \sigma, out \rangle \rightarrow \langle \sigma'', out'' \rangle \qquad \rho \vdash \langle inst_2, \sigma'', out'' \rangle \rightarrow \langle \sigma', out' \rangle}{\rho \vdash \langle \texttt{seq}\ inst_1\ inst_2, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle} \quad (8)$$

$$\frac{\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle boolean(true), \sigma'', out'' \rangle \quad \rho \vdash \langle inst_1, \sigma'', out'' \rangle \rightarrow \langle \sigma', out' \rangle}{\rho \vdash \langle \texttt{if}\ exp\ inst_1\ inst_2, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle} \quad (9)$$

Sémantique des instructions II

$$\frac{\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle boolean(false), \sigma'', out'' \rangle \quad \rho \vdash \langle inst_2, \sigma'', out'' \rangle \rightarrow \langle \sigma', out' \rangle}{\rho \vdash \langle \texttt{if}\ exp\ inst_1\ inst_2, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle} \quad (10)$$

$$\frac{\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle boolean(true), \sigma'', out'' \rangle \quad \rho \vdash \langle inst, \sigma'', out'' \rangle \rightarrow \langle \sigma''', out''' \rangle \quad \rho \vdash \langle \texttt{while}\ exp\ inst, \sigma''', out''' \rangle \rightarrow \langle \sigma', out' \rangle}{\rho \vdash \langle \texttt{while}\ exp\ inst, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle} \quad (11)$$

Sémantique des instructions III

$$\frac{\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle boolean(false), \sigma', out' \rangle}{\rho \vdash \langle \texttt{while}\ exp\ inst, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle} \quad (12)$$

$$\frac{\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle v, \sigma', out' \rangle}{\rho \vdash \langle \texttt{assign}\ id\ exp, \sigma, out \rangle \rightarrow \langle \sigma'[\rho(id) \mapsto v], out' \rangle} \quad (13)$$

$$\frac{\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle v, \sigma', out' \rangle}{\rho \vdash \langle \texttt{writefield idself}\ id\ exp, \sigma, out \rangle \rightarrow \langle writeField(\rho(\texttt{idself}), id, v, \sigma'), out' \rangle} \quad (14)$$

Sémantique des instructions IV

$$\frac{\rho \vdash \langle exp_r, \sigma, out \rangle \rightarrow \langle v_r, \sigma'', out'' \rangle \quad \rho \vdash \langle exp, \sigma'', out'' \rangle \rightarrow \langle v, \sigma', out' \rangle}{\rho \vdash \langle \texttt{writefield}\ exp_r\ id\ \text{exp}, \sigma, out \rangle \rightarrow \langle writeField(v_r, id, v, \sigma'), out' \rangle}\ exp_r \neq \texttt{idself} \wedge v_r \in Oid \quad (15)$$

$$\frac{\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle v, \sigma', out' \rangle}{\rho \vdash \langle \texttt{writeln}\ exp, \sigma, out \rangle \rightarrow \langle \sigma', out' + toString(v) \rangle}\ \neg v \in Oid \quad (16)$$

$$\frac{\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle v, \sigma', out' \rangle}{\rho \vdash \langle \texttt{writeln}\ exp, \sigma, out \rangle \rightarrow \langle \sigma', out' + toString(\sigma'(@(v))) \rangle}\ v \in Oid \quad (17)$$

Sémantique du corps de méthode I

- Dans le corps d'une méthode, la sémantique des instructions reste la même à ceci près qu'il faut gérer la valeur de retour, ce qui donne un relation $\rho \vdash \langle inst, \sigma, out\rangle \rightarrow \langle \sigma', out', r\rangle$:

$$\frac{\rho \vdash \langle inst_1, \sigma, out\rangle \rightarrow \langle \sigma'', out'', r_1\rangle \quad \rho \vdash \langle inst_2, \sigma'', out''\rangle \rightarrow \langle \sigma', out', r_2\rangle}{\rho \vdash \langle \texttt{seq}\ inst_1\ inst_2, \sigma, out\rangle \rightarrow \langle \sigma', out', r_2\rangle}\ undef(r_1) \tag{18}$$

$$\frac{\rho \vdash \langle inst_1, \sigma, out\rangle \rightarrow \langle \sigma', out', r_1\rangle}{\rho \vdash \langle \texttt{seq}\ inst_1\ inst_2, \sigma, out\rangle \rightarrow \langle \sigma', out', r_1\rangle}\ def(r_1) \tag{19}$$

Sémantique du corps de méthode II

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma'', out''\rangle \quad \rho \vdash \langle inst_1, \sigma'', out''\rangle \rightarrow \langle \sigma', out', r\rangle}{\rho \vdash \langle \texttt{if}\ exp\ inst_1\ inst_2, \sigma, out\rangle \rightarrow \langle \sigma', out', r\rangle} \tag{20}$$

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma'', out''\rangle \quad \rho \vdash \langle inst_2, \sigma'', out''\rangle \rightarrow \langle \sigma', out', r\rangle}{\rho \vdash \langle \texttt{if}\ exp\ inst_1\ inst_2, \sigma, out\rangle \rightarrow \langle \sigma', out', r\rangle} \tag{21}$$

Sémantique du corps de méthode III

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma'', out''\rangle \quad \rho \vdash \langle inst, \sigma'', out''\rangle \rightarrow \langle \sigma''', out''', r_1\rangle \quad \rho \vdash \langle \texttt{while}\ exp\ inst, \sigma''', out'''\rangle \rightarrow \langle \sigma', out', r_2\rangle}{\rho \vdash \langle \texttt{while}\ exp\ inst, \sigma, out\rangle \rightarrow \langle \sigma', out', r_2\rangle}\ undef(r_1) \tag{22}$$

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma'', out''\rangle \quad \rho \vdash \langle inst, \sigma'', out''\rangle \rightarrow \langle \sigma', out', r\rangle}{\rho \vdash \langle \texttt{while}\ exp\ inst, \sigma, out\rangle \rightarrow \langle \sigma', out', r\rangle}\ def(r) \tag{23}$$

Sémantique du corps de méthode IV

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle}{\rho \vdash \langle \texttt{while}\ exp\ inst, \sigma, out\rangle \rightarrow \langle \sigma', out', \varepsilon\rangle} \tag{24}$$

$$\frac{\rho \vdash \langle \texttt{assign}\ id\ exp, \sigma, out\rangle \rightarrow \langle \sigma', out'\rangle}{\rho \vdash \langle \texttt{assign}\ id\ exp, \sigma, out\rangle \rightarrow \langle \sigma', out', \varepsilon\rangle} \tag{25}$$

$$\frac{\rho \vdash \langle \texttt{writefield}\ exp_r\ id\ \text{exp}, \sigma, out\rangle \rightarrow \langle \sigma', out'\rangle}{\rho \vdash \langle \texttt{writefield}\ exp_r\ id\ \text{exp}, \sigma, out\rangle \rightarrow \langle \sigma', out', \varepsilon\rangle} \tag{26}$$

Sémantique du corps de méthode V

$$\frac{\rho \vdash \langle \texttt{writeln}\ exp, \sigma, out\rangle \rightarrow \langle \sigma', out'\rangle}{\rho \vdash \langle \texttt{writeln}\ exp, \sigma, out\rangle \rightarrow \langle \sigma', out', \varepsilon\rangle} \quad (27)$$

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle v, \sigma', out'\rangle}{\rho \vdash \langle \texttt{return}\ exp, \sigma, out\rangle \rightarrow \langle \sigma', out', v\rangle} \quad (28)$$

Sémantique des expressions - cas de base I

- Comme écrit précédemment, une expression peut modifier la mémoire et la sortie standard par les instructions et les expressions qu'elle utilise. Cela donne donc une relation $\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle V, \sigma', out'\rangle$:

$$\rho \vdash \langle \texttt{int}\ n, \sigma, out\rangle \rightarrow \langle integer(n), \sigma, out\rangle \quad (29)$$

$$\rho \vdash \langle \texttt{true}, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma, out\rangle \quad (30)$$

$$\rho \vdash \langle \texttt{false}, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma, out\rangle \quad (31)$$

$$\rho \vdash \langle id, \sigma, out\rangle \rightarrow \langle \sigma(\rho(id)), \sigma, out\rangle \quad (32)$$

Sémantique des expressions - arithmétiques I

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle integer(v_1), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle integer(v_2), \sigma', out'\rangle}{\rho \vdash \langle \texttt{plus}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle integer(v_1 + v_2), \sigma', out'\rangle} \quad (33)$$

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle integer(v_1), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle integer(v_2), \sigma', out'\rangle}{\rho \vdash \langle \texttt{minus}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle integer(v_1 - v_2), \sigma', out'\rangle} \quad (34)$$

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle integer(v_1), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle integer(v_2), \sigma', out'\rangle}{\rho \vdash \langle \texttt{times}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle integer(v_1 \times v_2), \sigma', out'\rangle} \quad (35)$$

Sémantique des expressions - `not`

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma', out''\rangle}{\rho \vdash \langle \texttt{not}\ exp, \sigma, out''\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle} \quad (36)$$

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle}{\rho \vdash \langle \texttt{not}\ exp, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle} \quad (37)$$

Sémantique des expressions - `and` I

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle}{\rho \vdash \langle \texttt{and}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle} \quad (38)$$

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle}{\rho \vdash \langle \texttt{and}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle} \quad (39)$$

Sémantique des expressions - `and` II

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle}{\rho \vdash \langle \texttt{and}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle} \quad (40)$$

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle}{\rho \vdash \langle \texttt{and}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle} \quad (41)$$

Sémantique des expressions - `or` I

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle}{\rho \vdash \langle \texttt{or}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle} \quad (42)$$

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle}{\rho \vdash \langle \texttt{or}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle} \quad (43)$$

Sémantique des expressions - `or` II

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle}{\rho \vdash \langle \texttt{or}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle} \quad (44)$$

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle}{\rho \vdash \langle \texttt{or}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle} \quad (45)$$

Sémantique des expressions - `less`

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle integer(v_1), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle integer(v_2), \sigma', out'\rangle}{\rho \vdash \langle \texttt{less}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle}\ v_1 < v_2 \qquad (46)$$

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle integer(v_1), \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle integer(v_2), \sigma', out'\rangle}{\rho \vdash \langle \texttt{less}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle}\ v_1 \geq v_2 \qquad (47)$$

Sémantique des expressions - `equal`

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle v_1, \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle v_2, \sigma', out'\rangle}{\rho \vdash \langle \texttt{equal}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle}\ v_1 = v_2 \qquad (48)$$

$$\frac{\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle v_1, \sigma'', out''\rangle \quad \rho \vdash \langle exp_2, \sigma'', out''\rangle \rightarrow \langle v_2, \sigma', out'\rangle}{\rho \vdash \langle \texttt{equal}\ exp_1\ exp_2, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle}\ v_1 \neq v_2 \qquad (49)$$

Sémantique des expressions - objets I

$$\rho \vdash \langle \texttt{nil}, \sigma, out\rangle \rightarrow \langle nil, \sigma, out\rangle \qquad (50)$$

$$\rho \vdash \langle \texttt{cexp}\ id, \sigma, out\rangle \rightarrow \langle id, \sigma, out\rangle \qquad isType(id, \rho) \qquad (51)$$

$$\frac{\rho \vdash \langle cexp, \sigma, out\rangle \rightarrow \langle id, \sigma, out\rangle}{\rho \vdash \langle \texttt{new}\ cexp, \sigma, out\rangle \rightarrow \langle oid, writeStore(a, object(oid, id, allocateFields(getOwnedFields(id, \rho))), \sigma), out\rangle} \qquad (52)$$

where $a = allocate(\sigma)$ *and* $oid = createOid(a)$

Sémantique des expressions - objets II

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle v, \sigma'', out''\rangle \quad \rho \vdash \langle cexp, \sigma'', out''\rangle \rightarrow \langle id, \sigma', out'\rangle}{\rho \vdash \langle \texttt{instanceOf}\ exp\ cexp, \sigma, out\rangle \rightarrow \langle boolean(true), \sigma', out'\rangle}\ v \in Oid \wedge inheritsFrom(classOf(v, \sigma'), id, \rho) \qquad (53)$$

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle v, \sigma'', out''\rangle \quad \rho \vdash \langle cexp, \sigma'', out''\rangle \rightarrow \langle id, \sigma', out'\rangle}{\rho \vdash \langle \texttt{instanceOf}\ exp\ cexp, \sigma, out\rangle \rightarrow \langle boolean(false), \sigma', out'\rangle}\ v \in Oid \wedge \neg inheritsFrom(classOf(v, \sigma'), id, \rho) \qquad (54)$$

Sémantique des expressions - `methodcall` I

$$\frac{\begin{array}{l}\rho \vdash \langle exp^*, \sigma, out\rangle \rightarrow \langle v^*, \sigma_1, out_1\rangle \\ \quad \rho_3 \vdash \langle inst, \sigma_3, out_1\rangle \rightarrow \langle \sigma_4, out_2, v\rangle \\ \qquad \text{where } id_c = classOf(\rho(\texttt{idself}), \sigma_1) \\ \qquad \text{and } method(id, id_1^*, id_2^*, inst) = lookForMethod(id_c, id_m, \rho) \\ \qquad \text{and } \rho_1 = \rho[\texttt{idsuper} \mapsto getSuper(id, \rho)] \\ \qquad \text{and } (\rho_2, \sigma_2) = storeAll(id_1^*, v^*, \rho_1, \sigma_1) \\ \qquad \text{and } (\rho_3, \sigma_3) = allocateAll(id_2^*, \rho_2, \sigma_2)\end{array}}{\rho \vdash \langle \texttt{methodcall idself } id_m\ exp^*, \sigma, out\rangle \rightarrow \langle v, \sigma_4, out_2\rangle} \quad (55)$$

Sémantique des expressions - `methodcall` II

$$\frac{\begin{array}{l}\rho \vdash \langle exp^*, \sigma, out\rangle \rightarrow \langle v^*, \sigma_1, out_1\rangle \\ \quad \rho_3 \vdash \langle inst, \sigma_3, out_1\rangle \rightarrow \langle \sigma_4, out_2, v\rangle \\ \qquad \text{where } id_c = \rho(\texttt{idsuper}) \\ \qquad \text{and } method(id, id_1^*, id_2^*, inst) = lookForMethod(id_c, id_m, \rho) \\ \qquad \text{and } \rho_1 = \rho[\texttt{idsuper} \mapsto getSuper(id, \rho)] \\ \qquad \text{and } (\rho_2, \sigma_2) = storeAll(id_1^*, v^*, \rho_1, \sigma_1) \\ \qquad \text{and } (\rho_3, \sigma_3) = allocateAll(id_2^*, \rho_2, \sigma_2)\end{array}}{\rho \vdash \langle \texttt{methodcall idsuper } id_m\ exp^*, \sigma, out\rangle \rightarrow \langle v, \sigma_4, out_2\rangle} \quad (56)$$

Sémantique des expressions - `methodcall` III

$$\frac{\begin{array}{l}\rho \vdash \langle exp_r, \sigma, out\rangle \rightarrow \langle v_r, \sigma_1, out_1\rangle \\ \quad \rho \vdash \langle exp^*, \sigma_1, out_1\rangle \rightarrow \langle v^*, \sigma_2, out_2\rangle \\ \qquad \rho_3 \vdash \langle inst, \sigma_4, out_2\rangle \rightarrow \langle \sigma_5, out_3, v\rangle \\ \qquad\quad \text{where } id_c = classOf(@v_r, \sigma_2) \\ \qquad\quad \text{and } method(id, id_1^*, id_2^*, inst) = lookForMethod(id_c, id_m, \rho) \\ \qquad\quad \text{and } \rho_1 = \rho[\texttt{idsuper} \mapsto getSuper(id, \rho)][\texttt{idself} \mapsto v_r] \\ \qquad\quad \text{and } (\rho_2, \sigma_3) = storeAll(id_1^*, v^*, \rho_1, \sigma_2) \\ \qquad\quad \text{and } (\rho_3, \sigma_4) = allocateAll(id_2^*, \rho_2, \sigma_3)\end{array}}{\rho \vdash \langle \texttt{methodcall } exp_r\ id_m\ exp^*, \sigma, out\rangle \rightarrow \langle v, \sigma_5, out_3\rangle}\ v_r \in Oid \quad (57)$$

où $@v_r$ est l'adresse correspondant à l'identifiant d'objet v_r.

Sémantique des expressions - `readfield`

$$\rho \vdash \langle \texttt{readField idself } id, \sigma, out\rangle \rightarrow \langle fieldsOf(\sigma(@(\rho(\texttt{idself}))))(id), \sigma, out\rangle \quad (58)$$

$$\frac{\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle v_r, \sigma', out'\rangle}{\rho \vdash \langle \texttt{readField } exp\ id, \sigma, out\rangle \rightarrow \langle fieldsOf(\sigma'(@v_r))(id), \sigma', out'\rangle}\ v_r \in Oid \quad (59)$$

Preuves de propriétés sémantiques à partir de SOS I

- Dans la même veine que la preuve de la propriété syntaxique sur le nombre d'opérateurs et d'opérandes vue précédemment, il est possible de prouver des propriétés sémantiques sur les programmes.
- À partir d'une SOS, on peut appliquer la technique de preuve par induction structurelle.

Théorème de complétude sur les expressions arithmétiques I

Lemme

Pour toute expression arithmétique exp de BOPL telle que tous les identifiants id apparaissant dans exp sont dans le domaine de ρ *et* $\rho(id)$ *est dans le domaine de* σ*, il existe un* $n \in \mathbb{Z}$ *tel que* $\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle n, \sigma, out \rangle$.

- Cas de base
  - Si $exp = \texttt{int}\, n$, alors n représente l'entier cherché.
  - Si $exp = id$; alors $n = \sigma(\rho(id))$ représente l'entier cherché.

Théorème de complétude sur les expressions arithmétiques II

- Cas inductifs
  - Si $exp = \texttt{plus}\ exp_1\ exp_2$ alors, par l'hypothèse d'induction on a $\rho \vdash \langle exp_1, \sigma, out \rangle \rightarrow \langle n_1, \sigma, out \rangle$ et $\rho \vdash \langle exp_2, \sigma, out \rangle \rightarrow \langle n_2, \sigma, out \rangle$. Par la règle d'inférence sur l'expression d'addition, $n_1 + n_2$ est l'entier cherché.
  - Les cas $exp = \texttt{minus}\ exp_1\ exp_2$ et $exp = \texttt{times}\ exp_1\ exp_2$ sont similaires.

Théorème de consistance sur les expressions arithmétiques I

Lemme

Pour toute expression arithmétique exp de BOPL telle que tous les identifiants id apparaissant dans exp sont dans le domaine de ρ *et* $\rho(id)$ *est dans le domaine de* σ*, si* $\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle n_1, \sigma, out\rangle$ *et* $\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle n_2, \sigma, out\rangle$*, alors* $n_1 = n_2$*.*

- Cas de base
  - Si $exp = \texttt{int}\, n$, alors les deux transitions doivent utiliser la même règle sur les constantes entières et donc la même règle doit donner le même résultat.

Théorème de consistance sur les expressions arithmétiques II

  - Si $exp = id$; alors les deux transitions doivent utiliser la même règle sur les identifiants et donc la même règle doit donner le même résultat $n_1 = n_2 = \sigma(\rho(id))$.
- Cas inductifs
  - Si $exp = \texttt{plus}\ exp_1\ exp_2$ alors, par l'application de la règle sur l'addition, on a pour $\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle n_1, \sigma, out\rangle$:

$$\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle m_1, \sigma, out\rangle$$

$$\rho \vdash \langle exp_2, \sigma, out\rangle \rightarrow \langle m_2, \sigma, out\rangle$$

et pour $\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle n_2, \sigma, out\rangle$:

$$\rho \vdash \langle exp_1, \sigma, out\rangle \rightarrow \langle k_1, \sigma, out\rangle$$

Théorème de consistance sur les expressions arithmétiques III

$$\rho \vdash \langle exp_2, \sigma, out\rangle \rightarrow \langle k_2, \sigma, out\rangle$$

Par l'hypothèse d'induction on a $m_1 = k_1$ et $m2 = k_2$ et ainsi $n_1 = m_1 + m_2 = k_1 + k_2 = n_2$.

- Les cas $exp = \texttt{minus}\ exp_1\ exp_2$ et $exp = \texttt{times}\ exp_1\ exp_2$ sont similaires.

Équivalence sémantique I

Théorème

Deux instructions $inst_1$ *et* $inst_2$ *sont sémantiquement équivalentes ssi pour tout* $\rho, \sigma, out, \sigma', out'$*, on a* $\rho \vdash \langle inst_1, \sigma, out\rangle \rightarrow \langle \sigma', out'\rangle$ *ssi* $\rho \vdash \langle inst_2, \sigma, out\rangle \rightarrow \langle \sigma', out'\rangle$

Lemme

Les instructions $\texttt{if}\ exp\ inst_1\ inst_2$ *et* $\texttt{if}\ (\texttt{not}\ exp)\ inst_2\ inst_1$ *sont sémantiquement équivalentes.*

Équivalence sémantique II

- Cas 1 : si $\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle true, \sigma'', out'' \rangle$, alors par la règle d'inférence `if`, on a $\rho \vdash \langle \texttt{if}\ exp\ inst_1\ inst_2, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle$ si $\rho \vdash \langle inst_1, \sigma'', out'' \rangle \rightarrow \langle \sigma', out' \rangle$, alors qu'on a $\rho \vdash \langle \texttt{if}\ (\texttt{not}\ exp)\ inst_2\ inst_1, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle$ si $\rho \vdash \langle inst_1, \sigma'', out'' \rangle \rightarrow \langle \sigma', out' \rangle$.
- Cas 2 : si $\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle false, \sigma'', out'' \rangle$, alors par la règle d'inférence `if`, on a $\rho \vdash \langle \texttt{if}\ exp\ inst_1\ inst_2, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle$ si $\rho \vdash \langle inst_2, \sigma'', out'' \rangle \rightarrow \langle \sigma', out' \rangle$, alors qu'on a $\rho \vdash \langle \texttt{if}\ (\texttt{not}\ exp)\ inst_2\ inst_1, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle$ si $\rho \vdash \langle inst_2, \sigma'', out'' \rangle \rightarrow \langle \sigma', out' \rangle$.

Équivalence sémantique III

Note : cette preuve est incomplète si on considère le cas où les instructions peuvent ne pas terminer.

Principes I

- Les SOS, en particulier de type « *small-step* », sont beaucoup utilisées pour définir la sémantique des langages concurrents.
- En effet, elles permettent d'exprimer des comportements fins sur l'ordre d'exécution qui sont souvent au cœur des problématiques de mise en œuvre des langages concurrents.

Parallélisme d'exécution I

- Soit l'instruction parallèle || de syntaxe abstraite :

$$inst \quad ::= \quad inst_1 \;||\; inst_2$$

- On peut en exprimer la sémantique avec les règles d'inférences (simples) :

$$\frac{\langle inst_1, \sigma\rangle \;\rightarrow\; \langle skip, \sigma'\rangle}{\langle inst_1 \;||\; inst_2, \sigma\rangle \;\rightarrow\; \langle skip \;||\; inst_2, \sigma'\rangle}$$

$$\frac{\langle inst_2, \sigma\rangle \;\rightarrow\; \langle skip, \sigma'\rangle}{\langle inst_1 \;||\; inst_2, \sigma\rangle \;\rightarrow\; \langle inst_1 \;||\; skip, \sigma'\rangle}$$

Parallélisme d'exécution II

$$\frac{\langle inst_2, \sigma\rangle \;\rightarrow\; \langle skip, \sigma'\rangle}{\langle skip \;||\; inst_2, \sigma\rangle \;\rightarrow\; \langle skip, \sigma'\rangle}$$

$$\frac{\langle inst_1, \sigma\rangle \;\rightarrow\; \langle skip, \sigma'\rangle}{\langle inst_1 \;||\; skip, \sigma\rangle \;\rightarrow\; \langle skip, \sigma'\rangle}$$

- Cette sémantique utilise le non-déterminisme intrinsèque dans le choix des règles d'inférence pour exprimer le fait que les deux instructions mises en parallèle peuvent être exécutées dans n'importe quel ordre. Cela correspond à une sémantique du parallélisme par tous les entrelacements des sous-instructions.

Activités complémentaires I

- Regarder le texte de Plotkin sur les SOS.
- Comprendre l'implantation Prolog de la SOS de BOPL.