Analyse des Programmes et Sémantique (3)

Prof[r] Jacques Malenfant

(cours de J.Malenfant modifié et enseigné en 2013 par Basile Starynkévitch)

janvier-avril 2013

MI030 - APS

Cours (en M1) du Professeur Jacques Malenfant
http://pagesperso-systeme.lip6.fr/Jacques.Malenfant/ Professeur en informatique au Laboratoire d'Informatique de Paris 6

Enseigné en 2013 par Basile Starynkevitch
http://starynkevitch.net/Basile/
basile@starynkevitch.net & basile.starynkevitch@cea.fr
ingénieur chercheur au CEA, LIST - http://www-list.cea.fr/
travaille sur gcc-melt.org

♠ Nota Bene

Les transparents à fond rose (pages numérotées ♠) et les mots ♠ signalés ainsi ♣ sont de Basile Starynkevitch (dont les opinions n'engagent que lui) ♠

La plupart des transparents sont [recopiés de ceux] de J.Malenfant 2012, que je remercie. Ces transparents sont disponibles sous http://starynkevitch.net/Basile/

Rappels

On a vu dans le cours précédent une introduction aux **sémantiques opérationnelles structurelles** des languages de programmation :

- état de la machine (“virtuelle” ou “abstraite”) à un instant donné
- transitions entre états
- l’exécution d’une instruction modifie l’état, donc suit une transition

Deux approches complémentaires :

- sémantique à petits pas (“small step”) : étapes à faible niveau d’abstraction, proche des instructions élémentaires de la machine ; appropriée pour exprimer finement l’ordre d’exécution de ces instructions élémentaires
- sémantique à grand pas (“big step”) : transition plutôt globale

`http://en.wikipedia.org/wiki/Operational_semantics`

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 $\#rators(e) < \#rands(e)$.

- 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 :

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

- On peut donc appliquer la technique d'induction structurelle pour prouver le lemme précedent qui ne porte que sur la syntaxe du programme.

Preuve par induction structurelle II

Démonstration.

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 $0 = \#rators(e) < \#rands(e) >= 1$.
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 :

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

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

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 III

suite de la démonstration.

Induction ...

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.

♠ Les preuves sont généralement inductives, “calquées” sur la syntaxe *abstraite*. ♣

la syntaxe ne suffit pas

Le fait que `+` désigne l'addition et `*` la multiplication est conventionel. On pourrait échanger leur rôle (`+` devenant la multiplication et `*` devenant l'addition), et garder la *même* syntaxe abstraite !

De même, deux langages de programmation (par exemple un sous-ensemble de *C* et un sous-ensemble de *Pascal*) peuvent avoir une syntaxe différente, et [presque] la même sémantique. Changer les mots-clés (d'anglais `while` en français `tantque` ....) modifie peu un langage de programmation.

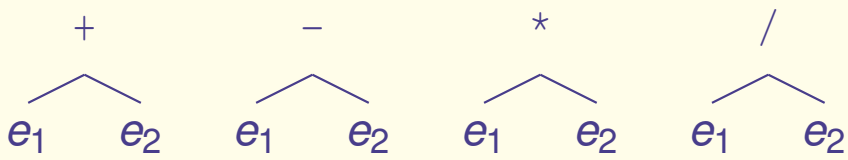
La sémantique est plus importante que la syntaxe.

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.
- Pour définir la sémantique, il faut avoir la syntaxe abstraite du langage :

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

♠ syntaxe abtraite des arbres



- 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 $\mathbb{Z}$ ou des réels $\mathbb{R}$, 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 $\Rightarrow_{\mathbb{R}} : \mathbb{Z} \rightarrow \mathbb{R}$

Règles d'inférences pour la sémantique

- Les règles d'inférence vont servir à produire le résultat du programme.
- Il faut donc définir une relation d'évaluation, notée comme précédemment $\rightarrow$. Les règles permettent de raisonner (ou enchaîner) sur des étapes de la forme : $e \rightarrow \mathbb{Z} \oplus \mathbb{R}$
- Elles sont définies également pour chaque construction de la grammaire abstraite.

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

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

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

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

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

- **etc** ... (pour les autres opérateurs)

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) :

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

♠ On **abstrait** (on oublie) le fait que les entiers machines soient bornés, et que les flottants ne soient pas les réels ; une sémantique plus proche de la machine serait plus compliquée à formaliser... (par exemple : l'addition des flottants n'est pas associative !) ♣

Introduction des variables

- 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 :

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

- Pour évaluer les références aux variables, il faut savoir récupérer la valeur associée à chacune des variables dans un environnement. Notons $\rho : Id \rightarrow \mathbb{Z} \oplus \mathbb{R}$ 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 :

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

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

$$\rho \vdash id \rightarrow \rho(id)$$

- 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.

Si on avait une affectation ...

Remarque informelle

Si on étendait le language avec une affectation [a] (qui en tant qu'expression vaut la valeur affectée) et un operateur d'évaluation séquentielle [b] $e ::= .... | \texttt{id} \leftarrow e | e, e$ l'environnement des variables ne serait plus dans le contexte, mais dans l'état.... et l'évaluation renverrait une valeur et un (nouvel) état ...

Il faudrait aussi définir l'ordre des affectations...

---

a. Comme en *C* ou en *Scheme* l'affectation serait une expression !
b. Comme en *C* l'operateur `,` ou en *Ocaml* l'opérateur `;` ou le **`begin`** de *Scheme*

Principes généraux

- 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 sémantique d'une expression `new` doit résulter dans un objet, qui doit donc aussi être représenté.
- La sémantique de l'appel de méthode impose de gérer le lien entre les objets et leur classe d'instantiation, pour déterminer la méthode à utiliser.
- L'existence d'expression `self` et `super` impose également la bonne définition de l'objet courant (`self`) et la classe de déclaration de la méthode courante (`super`).
- 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

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*).

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

- 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 $def : V \rightarrow T$ et $undef : V \rightarrow T$.

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.
- 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 :
  - $\rho : \mathit{Id} \rightarrow \mathit{Address}$, c'est-à-dire que pour obtenir l'adresse liée à un identifiant *id*, il suffit de faire $\rho(\mathit{id})$. ♠ C'est bien une adresse, pas une valeur ! ♣

Pourquoi trois éléments dans le contexte ? II

- La **mémoire** (« *store* ») peut donc être définie comme une fonction :
  - σ : *Address* $\rightarrow$ *V*, c'est-à-dire que pour obtenir la valeur associée à une adresse *a*, il suffit de faire $\sigma(a)$.
  - Quelques fonctions sont introduites pour compléter l'utilisation de la mémoire :
    - *allocate* : $\sigma \rightarrow$ *Address* permet d'allouer un nouvel espace mémoire et retourne l'adresse de ce dernier.
    - *createOid* : *Address* $\rightarrow$ *Oid* permet de créer un identifiant d'objet à partir de l'adresse à laquelle l'objet correspondant va être mémorisé.
    - @ : *Oid* $\rightarrow$ *Address* permet de retrouver l'adresse correspondant à un identifiant d'objets.
    - *allocateAll* : $Id^* \times \rho \times \sigma \rightarrow \rho \times \sigma$ alloue les variables et retourne les environnements et mémoires après allocations.
    - *storeAll* : $Id^* \times V^* \times \rho \times \sigma \rightarrow \rho \times \sigma$ alloue les variables et les initialise avec les valeurs passées en argument.
- 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 caractères. L'opération de concaténation des chaînes, notée $+$, sera utilisée pour accumuler les chaînes dans le flux de sortie.

A propos de la mémoire

Quelques remarques :

- la mémoire est supposée infinie, et on ne se préoccupe pas d'épuiser la resource correspondante
- la mémoire n'est pas [explicitement] libérée
- pas de "pile" ni de cadre d'appels explicites ("call stack", "call frame")
  technologiquement, la pile est importante dans les processeurs actuels
- implicitement, on voudrait un ramasse-miettes
  cf article "Garbage Collection" sur wikipedia
  (Cf A.Appel *Garbage collection can be faster than stack allocation*)

Gestion du contexte

- Contexte et sémantique des instructions : les instructions peuvent modifier la mémoire et le flux de sortie (`writeln`) mais pas l'environnement.
  ♠ l'environnement associe une addresse à chaque identificateur ♣

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

Représentation des classes II

- 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 :

$$\rho : \mathit{Id} \rightarrow \mathit{Classes} \oplus \mathit{Address}$$

- Quelques fonctions d'accès sont définies pour faciliter la manipulation des classes dans les règles :
  - *lookup* : $\mathit{Id} \times \mathit{Id} \times \rho \rightarrow \mathit{Method}$ retourne la méthode ♠ id_2 ♣ associée à une classe ♠ id_1 ♣.
  - *getSuper* : $\mathit{Id} \times \rho \rightarrow \mathit{Id}$ retourne l'identifiant de la superclasse d'une classe.
  - *inheritsFrom* : $\mathit{Id} \times \mathit{Id} \times \rho \rightarrow T$ retourne vrai si une classe (premier identifiant) hérite directement ou indirectement de la seconde (second identifiant).
  - *inheritedFields* : $\mathit{Id} \times \rho \rightarrow \mathit{Id}^*$ retourne la liste des identifiants des champs hérités par la classe.
  - *getOwnedFields* : $\mathit{Id} \times \rho \rightarrow \mathit{Id}^*$ donne les champs déclarés et hérités par une classe.

Représentation des objets

- 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
  - $fields : id \rightarrow V$ donne les valeurs associées aux champs par l'objet.
- Quelques fonctions sont définies pour faciliter la manipulation des objets :
  - $classOf : Oid \times \rho \rightarrow Id$ retourne l'identifiant de la classe d'instantiation d'un objet ;
  - $allocateFields : Id^* \rightarrow Id \rightarrow V$ crée la fonction *fields* initiale à partir des identifiants des champs ;
  - $fieldsOf : Object \rightarrow Id \rightarrow V$ retourne l'association des champs à leurs valeurs d'un objet.
  - $writeField : Oid \times Id \times v \times \sigma \rightarrow \sigma$ 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 :

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

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.

dictionnaire des méthodes

Selon les languages, les méthodes et leur invocation ont des status variés :

- dans certains languages (C++, Java, BOPL, Ocaml, ...), l'ensemble des méthodes invocables pour une classe donnée est figé
- dans d'autres languages (Javascript, Smalltalk, Self, ...) on peut ajouter/ôter dynamiquement une méthode

Ça correspond à des styles de programmation très différents, et l'invocation a un coût différent (indirection $\neq$ hash-table dictionnaire de méthode)

observation empirique : en un site d'invocation donné [un point du code source], la variabilité des invocations est généralement faible

Gestion du `self` et du `super`

- 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.
- 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 :

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

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 *program* $\rightarrow$ *out* définie par les règles :

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

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

Sémantique des déclarations de classes

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

$$\frac{\langle \uparrow class^*, \rho \rangle \rightarrow \rho'' \qquad \langle \downarrow class^*, \rho'' \rangle \rightarrow \rho'}{\langle class^*, \rho \rangle \rightarrow \rho'} \tag{3}$$

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

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

Sémantique des déclarations de méthodes

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

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

- 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 :

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

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

Sémantique des déclarations de variables locales

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

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

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 $\rho \vdash \langle inst, \sigma, out \rangle \rightarrow \langle \sigma', out' \rangle$:

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

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

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

Sémantique des instructions II

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

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

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

Sémantique des instructions III

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

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

Sémantique des instructions IV

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

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

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 $\rho \vdash \langle inst, \sigma, out \rangle \rightarrow \langle \sigma', out', r \rangle$:

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

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

Sémantique du corps de méthode II

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

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

Sémantique du corps de méthode III

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

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

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

Sémantique du corps de méthode III

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

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

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

Sémantique du corps de méthode IV

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

Sémantique des expressions - cas de base

- 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 $\rho \vdash \langle exp, \sigma, out \rangle \rightarrow \langle V, \sigma', out' \rangle$:

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

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

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

$$\rho \vdash \langle id, \sigma, out \rangle \rightarrow \langle \sigma(\rho(id)), \sigma, out \rangle \tag{32}$$

Sémantique des expressions - arithmétiques

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

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

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

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`

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

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

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

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

Sémantique des expressions - `or`

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

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

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

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

Sémantique des expressions - `less`

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

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

Sémantique des expressions - `equal`

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

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

Sémantique des expressions - objets

$$\rho \vdash \langle \texttt{nil}, \sigma, out \rangle \rightarrow \langle nil, \sigma, out \rangle \tag{50}$$

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

$$\frac{\rho \vdash \langle cexp, \sigma, out \rangle \rightarrow \langle id, \sigma, out \rangle}{\begin{array}{l}\rho \vdash \langle \texttt{new}\ cexp, \sigma, out \rangle \rightarrow \\ \quad \langle oid, writeStore(a, object(oid, id, \\ \qquad allocateFields(getOwnedFields(id, \rho))), \sigma), out \rangle \\ \quad \textit{where } a{=}allocate(\sigma) \textit{ and } oid{=}createOid(a)\end{array}} \tag{52}$$

Sémantique des expressions - `instanceOf`

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

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

Sémantique des expressions - `methodcall` I

♠ cas particuliers pour receveur **`self`** et **`super`** ♣

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

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

Sémantique des expressions - `methodcall` II

♠ cas général ♣

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

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

Sémantique des expressions - `readfield`

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

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

Preuves de propriétés sémantiques à partir de SOS

- 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

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.
- 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

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.
  - 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 $n_1 = n_2 = \sigma(\rho(id))$.

Th. consistance sur les expressions arithmétiques (suite)

- **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 $\rho \vdash \langle exp, \sigma, out\rangle \rightarrow \langle n_2, \sigma, out\rangle$:

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

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

    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

Definition : équivalence sémantique d'instructions

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 $\rho \vdash \langle inst_2, \sigma, out\rangle \rightarrow \langle \sigma', out'\rangle$

♠ On peut donc "exécuter" $inst_1$ ou $inst_2$ sans que ça change "quelque chose". Les compilateurs utilisent cette notion pour optimiser. ♣

Équivalence sémantique II

Lemme

Les instructions `if` *exp* $inst_1$ $inst_2$ *et* `if` (`not` *exp*) $inst_2$ $inst_1$ *sont sémantiquement équivalentes.*

- 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 $\rho \vdash \langle$`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 $\rho \vdash \langle$`if` (`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 $\rho \vdash \langle$`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 $\rho \vdash \langle$`if` (`not` *exp*) $inst_2$ $inst_1, \sigma, out\rangle \rightarrow \langle \sigma', out'\rangle$ si $\rho \vdash \langle inst_2, \sigma'', out''\rangle \rightarrow \langle \sigma', out'\rangle$.

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

Principes

- 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

- 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) :

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

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

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

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

- 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

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