Analyse des Programmes et Sémantique (2)

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/

apéritif : indécidabilité du problème de l'arrêt

http://fr.wikipedia.org/wiki/Problème_de_l'arrêt

Énoncé : **problème de l'arrêt**

Soit un programme P dont on a le code source, qui reçoit en entrée E. Décider, en examinant ce code source, si le programme P, avec l'entrée E, s'arrête en temps fini.

Existe-t-il un **programme** A prenant P et E comme entrées qui répond toujours correctement (répond `vrai` si le programme P s'arrête avec l'entrée E, et `faux` sinon) ?

programme qui s'arrête

```
int x = 10;
while (x > 0)
  x = x/2;
```

programme qui boucle

```
int x = 0;
while (true)
  x++;
```

Un programme peut être représenté par une donnée, son code source.

Non, **un tel programme** A **n'existe pas**. Par l'absurde : supposons qu'on ait un tel A.
Considerons le programme
$A'(C) =$ **si** $A(C,C)$ **alors boucler indéfiniment sinon terminer**.
Contradiction pour $A(A',A')$ donc A ne peut pas être.

Cours 2

Analyses statiques et vérification des types

Propriétés dépendantes du contexte

- Les grammaires indépendantes du contexte ne capturent qu'une partie des règles qui font qu'un programme est correct.
- En particulier, des propriétés comme le fait que les variables sont déclarées avant d'être utilisées, par exemple, ne peuvent s'exprimer dans une grammaire indépendante du contexte.
  - Une expression utilisant une variable serait correcte si elle se trouve dans une partie du programme où se trouve une déclaration de cette variable, mais pas si elle se trouve en dehors d'une telle partie.
  - *La validité de cette expression dépendrait donc du contexte dans lequel elle apparaît.*
- C'est le cas également de la *vérification des types*, qui consiste à s'assurer de la conformité des types dans toutes les expressions et instructions du programme.
- Pour s'attaquer à ce type de propriétés contextuelles, il faudrait passer à des grammaires *dépendantes* du contexte, mais les analyseurs pour ces grammaires sont trop gourmands en ressources (temps, espace) pour les utilisations concrètes (compilateurs, par exemple).

Définition des analyses statiques

- Pour définir les analyses statiques, il faut les exprimer dans un formalisme précis.
- Parmi les nombreux formalismes utilisables, nous allons dans un premier temps utiliser celui des *règles d'inférence*.

Définition : **Règle d'inférence**

Une règle d'inférence consiste en une conclusion déduite de prémisses, éventuellement sous le contrôle d'une condition. Sa forme générale est :

$$\frac{\textit{prémisse}_1 \quad \textit{prémisse}_2 \quad \ldots \quad \textit{prémisse}_n}{\textit{conclusion}} \ \textit{condition}$$

Lorsqu'une conclusion est vraie sans prémisse, que ce soit avec une condition ou non, on dit qu'il s'agit d'un axiome, donné sans utiliser la barre horizontale :

$$\textit{axiome} \qquad \textit{condition}$$

♠ on écrit parfois les différentes $\textit{prémisse}_i$ "en escalier" par manque de place ! ♣

Déduction naturelle

- Ces règles d'inférence sont apparues dans une forme de logique appelée *déduction naturelle*.
- Elles permettent de définir un *système de déduction* pour raisonner sur des *formules logiques*.
- Par exemple, trois règles d'inférence permettent d'exprimer les propriétés du connecteur logique *et* ($\wedge$) :

$$\frac{p \wedge q}{p} \qquad \frac{p \wedge q}{q} \qquad \frac{p \quad q}{p \wedge q}$$

- Une analyse statique exprimée dans ce formalisme construit donc un système de déduction pour la propriété recherchée.

Vérification des types dans les expressions arithmétiques

Définition : Vérification de types

Vérification statique de la bonne utilisation des types dans les constructions du langage, de manière à apporter des garanties sur l'absence d'erreurs de typage lors de l'exécution du programme.

- Il s'agit donc, pour chaque construction, de connaître le type des valeurs utilisées et de vérifier si ces types sont conformes à ceux des paramètres des opérations dans lesquelles elles sont utilisées.
- Par exemple, l'opération d'addition prend des valeurs de type numérique (entier, réel, ...), alors que l'opération de conjonction logique prend des valeurs de type booléen.
- Les règles de typage dépendent de la notion de conformité qui est appliquée.
  - La plus simple est d'exiger que les types soient les mêmes, mais des conformités plus souples existent.
  - Par exemple, la plupart des langage permettent de mélanger les types numériques en permettant la promotion des valeurs selon la chaine d'inclusion des ensembles de nombres (entier vers réel, par exemple).

Pourquoi un typage statique ?

1. Les ordinateurs actuels manipulent des données typées : les entiers et les flottants *IEEE-754* sont stockés dans des registres différents par des instructions machines distinctes ; la machine ne peut pas traiter un flottant comme un entier (sans conversion par une instruction explicite).
2. Les programmeurs ont besoin de comprendre le type des valeurs manipulées pour comprendre ce que fait un programme
3. C'est encore plus vrai avec des valeurs fonctionnelles
4. Certains languages [1] sont dynamiquement typés : chaque valeur contient un descripteur de type

1. Python, Javascript, et partiellement Java pour les objets, dont le descripteur de classe est un champ implicite ; en C++, les classes avec `virtual` ont des objets avec une v-table.

Les expressions arithmétiques simples

- Pour définir l'analyse, il faut avoir la syntaxe abstraite du langage. Considérons :

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

- Le résultat de l'analyse pourraît n'être que *oui* ou *non*, c'est-à-dire le programme est correctement typé ou non. En pratique, on va plutôt produire un nouveau **programme annoté** par les types déduits pendant l'analyse. Pour représenter ce programme annoté, nous utilisons une nouvelle grammaire abstraite :

$$\begin{aligned} te \quad &::= \quad te + te : ty \mid te - te : ty \mid te * te : ty \mid te \,/\, te : ty \\ &\qquad \mid \texttt{int} : \textsc{int} \mid \texttt{real} : \textsc{real} \\ ty \quad &::= \quad \textsc{int} \mid \textsc{real} \end{aligned}$$

- Notez ici les changements de polices de caractères destinés à faire apparaître clairement les distinctions entre les symboles terminaux de la grammaire abstraite : par exemple, `int` qui représente une valeur entière, et INT qui représente le type entier.

Annotations sémantiques en pratique

Les annotations sémantiques donnent *beaucoup* d'informations supplémentaires
(par exemple, une annotation de type pour chaque nœud de l'arbre de syntaxe abstrait)

Pour éviter de noyer l'utilisateur avec autant d'annotations, il faut en pratique les afficher à la demande (paresseusement) :

- bulles ou sous-fenêtres d'aide dans *Eclipse*
- avec `ocamlc -annot` et le mode *Tuareg* d'Emacs, `C-c C-t`
- etc...

L'interface utilisateur est donc importante dans les outils d'analyse statique, et ceux ci peuvent être gourmands en ressources.

En interne, les arbres sémantiques sont lourdement décorés. Informatiquement, les analyseurs sémantiques produisent des arbres différents (avec une structure de données différente) ou mantiennent les annotations "ailleurs" (tables de hash, ...)

Règles d'inférences pour le typage

- Les règles d'inférence vont servir à produire une traduction des programmes non-annotés vers les programmes annotés par les types.
- Il faut donc définir une **relation de traduction**, notée ici $\rightarrow$. Les règles permettent donc de raisonner (ici pour enchaîner) sur des étapes de traduction [2] de la forme :

$$e \rightarrow te$$

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

$$\texttt{int} \rightarrow \texttt{int} : \text{INT} \qquad \texttt{real} \rightarrow \texttt{real} : \text{REAL}$$

$$\frac{e_1 \rightarrow te_1 \qquad e_2 \rightarrow te_2}{e_1 + e_2 \rightarrow te_1 + te_2 : \text{INT}} \; typeOf(te_1) = typeOf(te_2) = \text{INT}$$

$$\frac{e_1 \rightarrow te_1 \qquad e_2 \rightarrow te_2}{e_1 + e_2 \rightarrow te_1 + te_2 : \text{REAL}} \; \neg(typeOf(te_1) = typeOf(te_2) = \text{INT})$$

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

2. ♠ traduction d'une syntaxe abstraite e en un "arbre typé" te ♣

Typage d'une expression

- Le typage d'une expression se fait en chaînant les règles d'inférence sous la forme d'un *arbre d'inférence*.[3]
- 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) :

$$\frac{\dfrac{5 \rightarrow 5:\mathsf{INT} \qquad 3 \rightarrow 3:\mathsf{INT}}{(5+3) \rightarrow (5:\mathsf{INT}+3:\mathsf{INT}):\mathsf{INT}} \qquad 2,5 \rightarrow 2,5:\mathsf{REAL}}{(5+3)*2,5 \rightarrow ((5:\mathsf{INT}+3:\mathsf{INT}):\mathsf{INT}*2,5:\mathsf{REAL}):\mathsf{REAL}}$$

3. ♠ Cet arbre d'inférence décrit le "raisonnement" ou le "calcul" aboutissant à l' "arbre typé".

règles d'inférence et leur contexte

En pratique, les règles d'inférence vont être progressivement étoffées (par un contexte de plus en plus lourd), avec :

- l'environment de typage des variables Γ
- la relation Δ d'héritage entre classes
- la classe courante δ
- le type de retour courant τ

Voir les livres de Benjamin C. Pierce dont *Types and Programming Languages*

Introduction des variables I

- Les expressions arithmétiques précédentes ne comportent pas de variables. Que se passe-t-il si on en ajoute ?
  La grammaire abstraite :

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

  La grammaire abstraite annotée :

$$\begin{aligned} te \quad ::= \quad & te + te : ty \mid te - te : ty \mid te * te : ty \mid te / te : ty \mid \\ & \texttt{int} : \text{INT} \mid \texttt{real} : \text{REAL} \mid id : ty \\ ty \quad ::= \quad & \text{INT} \mid \text{REAL} \end{aligned}$$

- Pour vérifier le typage des variables, il faut :
  1. savoir récupérer le type associé à chacune des variables, ce qui peut venir de déclarations précédentes dans un programme ;
  2. vérifier que ce type est conforme au type attendu par l'opération qui s'applique sur la valeur de cette variable.

Introduction des variables II

- Récupérer le type de la variable peut se faire par un **environnement de typage**. Notons $\Gamma : id \rightarrow ty$ la fonction associant à chaque variable le type qui lui a été déclaré.
- Les règles d'inférence vont maintenant être écrites pour une relation de traduction *dans le contexte d'un environnement de typage* de la forme :

$$\Gamma \vdash e \rightarrow te$$

- Pour les règles précédentes cela donne :

$$\Gamma \vdash \texttt{int} \rightarrow \texttt{int} : \text{INT} \qquad \Gamma \vdash \texttt{real} \rightarrow \texttt{real} : \text{REAL}$$

$$\frac{\Gamma \vdash e_1 \rightarrow te_1 \qquad \Gamma \vdash e_2 \rightarrow te_2}{\Gamma \vdash e_1 + e_2 \rightarrow te_1 + te_2 : \text{INT}} \; typeOf(te_1) = typeOf(te_2) = \text{INT}$$

$$\frac{\Gamma \vdash e_1 \rightarrow te_1 \qquad \Gamma \vdash e_2 \rightarrow te_2}{\Gamma \vdash e_1 + e_2 \rightarrow te_1 + te_2 : \text{REAL}} \; \neg(typeOf(te_1) = typeOf(te_2) = \text{INT})$$

- Et pour le cas des variables, on ajoute la règle :

$$\Gamma \vdash id \rightarrow id : \Gamma(id)$$

Introduction des variables III

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

♠ typage dans les langages de programation

Le typage sert partout :

- dans tous les compilateurs, pour générer les bonnes instructions (flottant $\neq$ entier $\neq$ long $\neq$ pointeur)
- en *C*, arithmétique (des nombres, des pointers, ...) si on a `float x, y;` l'addition `x+y` se fait en `float` (C99) ou en `double` (C90) selon le standard !
- en Java 5 et C++99 : pour les génériques et les `template`s
- en Ocaml ou Haskell, **inférence de type** (ou "synthèse de type") on peut y annoter explicitement le type de certaines expressions.
- en *C++2011* `auto x = 2;` compris comme `signed int x = 2;` en pratique, très utile avec les itérateurs, avec `std::vector<int> vec;` simplement `auto it = vec.begin();` au lieu de `std::vector<int>::iterator it = vec.begin();`

Le système de types de BOPL

- Le système de type pour BOPL est fondé sur l'utilisation des noms de classes comme types.
- Hormis les types définis par les classes du programmeur, il existe quatre types prédéfinis :

  `id Object` : la racine de l'arbre d'héritage ;
  `id Int` : les valeurs entières ;
  `id Bool` : les valeurs booléennes ; et
  `id Void` : l'absence de valeur.

- Les méthodes vont devoir être typées, ce que nous allons faire avec un type *construit* de la forme suivante :

$$FormalType_1 \times \ldots \times FormalType_n \rightarrow ReturnType$$

♠ On travaille donc dans un "domaine de types" avec leur propre "syntaxe" et leur "arbres abstraits" (i.e. leur représentation en machine) ; il y a une infinité de types. ♣

Les déclarations de classes

- Un programme BOPL comporte deux grandes parties :
  1. les déclarations de classes, et
  2. le corps du programme, éventuellement débuté par des déclarations de variables locales.
- De la partie déclarations des classes, il nous faut donc obtenir :
  - les noms des classes avec leurs relations d'héritage pour savoir ensuite typer des choses comme le `super` ;
  - les noms des variables d'instance, classe par classe, avec le type de leur déclaration, pour pouvoir ensuite typer les accès à ces variables dans les expressions ;
  - les noms des méthodes, classe par classe, avec leur type, pour pouvoir ensuite typer correctement les appels de méthodes.
- Pour conserver ces informations, nous décidons de définir deux fonctions :

  $\Delta : id \to id$ définit la **relation d'héritage**, c'est-à-dire si $\Delta(B) = A$, alors B est sous-classe ♠ directe ♣ de A.

  $\Gamma : id \to t$ définit la **relation *type-de***, c'est-à-dire si $\Gamma(a) = T$, alors le type de a est T.

Le corps du programme

- Dans la partie corps du programme, on utilise les classes pour créer des objets, et sur ces objets on peut accéder aux variables d'instance et à leurs méthodes.
  - les expressions `new` et `instanceof` utilisent des noms de classes, ce qui donne les expressions de classes `cexp` dans la grammaire abstraite ;
  - il faut donc représenter dans le système de types le type de ces expressions ♠ de classe ♣ différemment du type des expressions dont le résultat est un objet instance de la classe en question ;
  - dans l'expression `new id Point` :
    - la sous-expression `id Point` aura pour type $aType($`id Point`$)$, c'est-à-dire la classe `Point`,
    - tandis que l'expression `new` aura pour type `id Point`, c'est-à-dire une instance de la classe `Point` ;
    - définissons les opérations $\Uparrow T = aType(T)$ et $\Downarrow aType(T) = T$ pour faire la médiation entre ces niveaux de types ;
  - pour les variables d'instance et les méthodes, il faut retenir leur classe d'appartenance, ce que nous allons faire en les introduisant dans l'environnement de typage sous la forme d'un tuple <*classe*>@<*id*>, donc le type de Γ est à proprement parler $(Id \oplus Id@Id) \rightarrow Id$.

Les déclarations de variables locales

- Que ce soit dans le corps du programme et dans le corps des méthodes, la syntaxe autorise la déclaration de variables locales.
- Ces variables avec leurs types vont devoir être introduites dans l'environnement de typage, comme nous l'avons vu dans le cas des expressions arithmétiques simples.
- Toujours comme dans le cas des expressions, il faut mémoriser les types de ces variables dans l'environnement de typage avant de vérifier les instructions qui sont dans la portée de ces déclarations.

Grammaire abstraite *typée* de BOPL I

- Comme dans le cas des expressions arithmétiques simples, la vérification n'aura pas seulement pour but de dire si le programme est correctement typé ou non ; elle va produire *une nouvelle version du programme annotée* par les types.
- La grammaire abstraite annotée suivante sera la cible de cette transformation.

$$\begin{aligned} \textit{tprogram} &::= \texttt{program}\ \textit{tclass}^*\ \textit{tvar}^*\ \textit{tinst}\ :\ \textit{type} \\ \textit{tclass} &::= \texttt{class}\ \textit{id}\ \textit{tcexp}\ \textit{tvar}^*\ \textit{tmethod}^*\ :\ \textit{type} \end{aligned}$$

$$\begin{aligned} \textit{tcexp} &::= \texttt{cexp}\ \textit{id}\ :\ \textit{type} \\ \textit{tmethod} &::= \texttt{method}\ \textit{id}\ \textit{tvar}^*\ \textit{id}\ \textit{tvar}^*\ \textit{tseq}\ :\ \textit{type} \\ \textit{tvar} &::= \texttt{var}\ \textit{tcexp}\ \textit{id}\ :\ \textit{type} \\ \textit{tinst} &::= \texttt{seq}\ \textit{tinst}\ \textit{tinst}\ :\ \textit{type} \mid \texttt{assign}\ \textit{id}\ \textit{texp}\ :\ \textit{type} \mid \\ &\quad \texttt{writefield}\ \textit{texp}\ \textit{id}\ \textit{texp}\ :\ \textit{type} \mid \\ &\quad \texttt{if}\ \textit{texp}\ \textit{tinst}\ \textit{tinst}\ :\ \textit{type} \mid \texttt{while}\ \textit{texp}\ \textit{tinst}\ :\ \textit{type} \mid \\ &\quad \texttt{return}\ \textit{texp}\ :\ \textit{type} \mid \texttt{writeln}\ \textit{texp}\ :\ \textit{type} \end{aligned}$$

Grammaire abstraite *typée* de BOPL II

♠ Expressions abstraites typées et types : ♣

$$
\begin{aligned}
\mathit{texp} \quad ::= \quad & \mathit{int} \ : \ \mathit{type} \mid \texttt{true} \ : \ \mathit{type} \mid \texttt{false} \ : \ \mathit{type} \mid \texttt{not} \ \mathit{texp} \ : \ \mathit{type} \mid \\
& \texttt{nil} \ : \ \mathit{type} \mid \texttt{self} \ : \ \mathit{type} \mid \texttt{super} \ : \ \mathit{type} \mid \\
& \texttt{new} \ \mathit{tcexp} \ : \ \mathit{type} \mid \texttt{instanceof} \ \mathit{exp} \ \mathit{exp} \ : \ \mathit{type} \mid \\
& \mathit{id} \ : \ \mathit{type} \mid \texttt{methodcall} \ \mathit{exp} \ \mathit{id} \ \mathit{exp}^* \ : \ \mathit{type} \mid \\
& \texttt{readfield} \ \mathit{exp} \ \mathit{id} \ : \ \mathit{type} \mid \texttt{plus} \ \mathit{exp} \ \mathit{exp} \ : \ \mathit{type} \mid \\
& \texttt{minus} \ \mathit{exp} \ \mathit{exp} \ : \ \mathit{type} \mid \texttt{times} \ \mathit{exp} \ \mathit{exp} \ : \ \mathit{type} \mid \\
& \texttt{equal} \ \mathit{exp} \ \mathit{exp} \ : \ \mathit{type} \mid \texttt{and} \ \mathit{exp} \ \mathit{exp} \ : \ \mathit{type} \mid \\
& \texttt{or} \ \mathit{exp} \ \mathit{exp} \ : \ \mathit{type} \mid \texttt{less} \ \mathit{exp} \ \mathit{exp} \ : \ \mathit{type}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{type} \quad ::= \quad & \mathit{id} \mid \mathit{aType}(\mathit{id}) \\
& \mid \texttt{id Bool} \mid \texttt{id Int} \mid \texttt{id Object} \mid \texttt{id Void}
\end{aligned}
$$

Encore quelques éléments de contexte : méthodes

- Les méthodes peuvent retourner un résultat grâce à l'instruction `return`.
- Le type de l'expression servant dans l'instruction `return` doit être conforme au type de retour déclaré dans la signature de la méthode.
- On connaît le type de retour dès le début de la vérification du type de la méthode, mais cette information doit être colportée jusqu'aux instructions `return` qui sont dans le corps de la méthode.
- On introduit donc un nouvel élément de contexte pour vérifier le type des instructions :

  $\tau : Id$ type de retour courant.

Encore quelques éléments de contexte : `self` et `super`

- Dans un langage à objets, on utilise `super` dans une expression du corps d'une méthode pour appeler la définition de la méthode masquée par celle-ci.
- Comme nous admettons la *contravariance* dans les types des paramètres *formels* et du type du *résultat*, vérifier correctement les types nécessite d'aller chercher cette méthode masquée pour récupérer son type.
- Pour cela, il faut savoir quelle est la classe courante dans laquelle nous sommes lors de la vérification des instructions du corps d'une méthode pour trouver sa superclasse.
- On introduit donc un nouvel élément de contexte pour vérifier le type des instruction :

  $\delta : Id$ identifiant de la classe courante.
- Cet élément de contexte permettra par ailleurs un traitement «*défensif*» du typage de `self` consistant à le voir comme du type de la classe courante, en se fiant aux règles de non-redéfinition des variables d'instance et de contravariance des méthodes.

Notations pour le traitement des listes

$\uparrow L$ premier de la liste L.

$\downarrow L$ reste de la liste.

$e \S L$ liste formée du premier élément e et d'un reste de liste L.

- La vérification et traduction d'un programme est donné par la relation :

$$\mathit{program} \rightarrow \mathit{tprogram}$$

- Vérifier un programme demande de vérifier les déclarations de classes, ce qui va donner la relation d'héritage Δ et l'environnement de typage Γ dans lesquels les instructions du corps du programme seront vérifiées.

$$\frac{\begin{array}{c}\langle \mathit{class}^*, \varnothing, \varnothing\rangle \rightarrow \langle \mathit{tclass}^*, \Delta, \Gamma\rangle \\ \Delta, \varnothing \vdash \langle \mathit{var}^*, \Gamma\rangle \rightarrow \langle \mathit{tvar}^*, \Gamma'\rangle \\ \Delta, \Gamma', \varnothing, \varnothing \vdash \mathit{inst} \rightarrow \mathit{tinst}\end{array}}{\texttt{program}\ \mathit{class}^*\ \mathit{var}^*\ \mathit{inst} \rightarrow (\texttt{program}\ \mathit{tclass}^*\ \mathit{tvar}^*\ \mathit{tinst}) : \texttt{id Void}} \quad (1)$$

Traitement des déclarations de variables locales I

- Les déclarations de variables introduisent dans l'environnement de typage les liaisons entre les identifiants de variables et leur type déclaré.
- Quand ce sont des variables d'instance, il faut préfixer le nom de la variable par le nom de sa classe (Class@Id).
- Par contre, elles ne modifient pas la relation d'héritage, donc cette dernière apparaît uniquement dans le contexte des règles, qui définissent donc les relations :

$$\Delta, \delta \vdash \langle var^*, \Gamma \rangle \rightarrow \langle tvar^*, \Gamma' \rangle$$

$$\Delta, \delta \vdash \langle var, \Gamma \rangle \rightarrow \langle tvar, \Gamma' \rangle$$

Traitement des déclarations de variables locales II

$$\frac{\Delta,\delta \vdash \langle \uparrow var^*, \Gamma\rangle \rightarrow \langle tvar, \Gamma''\rangle \qquad \Delta,\delta \vdash \langle \downarrow var^*, \Gamma''\rangle \rightarrow \langle tvar^*, \Gamma'\rangle}{\Delta,\delta \vdash \langle var^*, \Gamma\rangle \rightarrow \langle tvar \S tvar^*, \Gamma'\rangle} \quad (2)$$

$$\frac{\Delta,\Gamma,\varnothing \vdash cexp \rightarrow tcexp}{\begin{array}{l}\Delta,\varnothing \vdash \langle \texttt{var}\ cexp\ id, \Gamma\rangle \rightarrow \\ \quad \langle (\texttt{var}\ tcexp\ id :\Downarrow typeOf(tcexp)) : \texttt{id Void}, \\ \qquad \Gamma[\Downarrow typeOf(tcexp)/id]\rangle\end{array}} \quad (3)$$

$$\frac{\Delta,\Gamma,\varnothing \vdash cexp \rightarrow tcexp}{\begin{array}{l}\Delta,\delta \vdash \langle \texttt{var}\ cexp\ id, \Gamma\rangle \rightarrow \\ \quad \langle (\texttt{var}\ tcexp\ id :\Downarrow typeOf(tcexp)) : \texttt{id Void}, \\ \qquad \Gamma[\Downarrow typeOf(tcexp)/\delta@id]\rangle\end{array}} \ \delta \neq \varnothing \quad (4)$$

Vérification des déclarations de classes

- Comme il est possible de le voir dans les règles précédentes, la vérification des déclarations de classes modifie la relation d'héritage et l'environnement de typage. C'est ainsi que ces deux éléments deviennent parties prenantes de la relation de vérification/traduction qui prend maintenant un triplet et rend un triplet :

$$\langle class, \Delta, \Gamma \rangle \rightarrow \langle tclass, \Delta', \Gamma' \rangle$$

- Le traitement des listes de déclarations de classes demande simplement d'enfiler correctement les Δ et Γ :

$$\frac{\langle \uparrow class^*, \Delta, \Gamma \rangle \rightarrow \langle tclass, \Delta'', \Gamma'' \rangle \quad \langle \downarrow class^*, \Delta'', \Gamma'' \rangle \rightarrow \langle tclass^*, \Delta', \Gamma' \rangle}{\langle class^*, \Delta, \Gamma \rangle \rightarrow \langle tclass \S tclass^*, \Delta', \Gamma' \rangle} \quad (5)$$

Vérification de chaque déclaration de classe I

- Pour une classe, il y a trois éléments à traiter : l'expression donnant la superclasse, les déclarations de variables d'instance et les déclarations de méthodes.
- Notons que le traitement des variables d'instance est différent de celui des variables locales, dans la mesure où elles introduisent une entrée dans l'environnement de typage qui doit être visible sur l'ensemble du programme alors que les variables locales ne le font que pour la partie du programme où elles sont visibles.
  Les variables locales ont une portée limitée, alors que les variables d'instances ont une portée illimitée.
- Les déclarations de méthodes introduisent les types de ces dernières dans l'environnement de typage.

Vérification de chaque déclaration de classe II

$$\frac{\begin{array}{l}\Delta, \Gamma, id \vdash cexp \rightarrow tcexp \\ \quad \Delta[\Downarrow typeOf(tcexp)/id], id \vdash \langle var^*, \Gamma \rangle \rightarrow \langle tvar^*, \Gamma'' \rangle \\ \quad\quad \Delta[\Downarrow typeOf(tcexp)/id], id \vdash \langle method^*, \Gamma'' \rangle \rightarrow \langle tmethod^*, \Gamma' \rangle\end{array}}{\begin{array}{l}\langle \texttt{class}\ id\ cexp\ var^*\ method^*, \Delta, \Gamma \rangle \rightarrow \\ \quad \langle (\texttt{tclass}\ id\ tcexp\ tvar^*\ tmethod^*) : \texttt{id Void}, \\ \quad\quad\quad \Delta[\Downarrow typeOf(tcexp)/id], \Gamma' \rangle\end{array}} \quad (6)$$

- Notez l'introduction d'héritage de la relation entre la nouvelle classe et sa superclasse dans Δ.

Vérification des déclarations de méthodes

- Pour la vérification des déclarations de méthodes, il est nécessaire d'avoir dans le contexte la classe courante pour typer correctement les expressions `super`, comme mentionné précédemment. On définit donc les relations :

$$\Delta, \delta \vdash \langle method^*, \Gamma \rangle \rightarrow \langle tmethod^*, \Gamma' \rangle$$

$$\Delta, \delta \vdash \langle method, \Gamma \rangle \rightarrow \langle tmethod, \Gamma' \rangle$$

- Comme pour les déclarations de variables, le traitement de la liste de déclarations de méthodes demandent surtout de bien enfiler les Δ et Γ :

$$\frac{\Delta, \delta \vdash \langle \uparrow method^*, \Gamma \rangle \rightarrow \langle tmethod, \Gamma'' \rangle \qquad \Delta, \delta \vdash \langle \downarrow method^*, \Gamma'' \rangle \rightarrow \langle tmethod^*, \Gamma' \rangle}{\Delta, \delta \vdash \langle method^*, \Gamma \rangle \rightarrow \langle tmethod \S tmethod^*, \Gamma' \rangle} \quad (7)$$

Vérification de chaque méthode

- Pour chaque méthode, l'objectif est d'introduire dans l'environnement de typage le type de la méthode, obtenu à partir des types de ses paramètres et du type de retour, puis de vérifier les types de son corps.
- Il faut aussi vérifier que si la nouvelle méthode redéfinit une méthode existante dans une superclasse, elle doit être contravariante par rapport à cette dernière.
- Notez que la vérification des types du corps introduit le type de retour dans le contexte pour vérifier les types des instructions `return`, comme mentionné précédemment.

$$\frac{\begin{array}{l}\Delta,\varnothing \vdash \langle var_1^*,\Gamma\rangle \rightarrow \langle tvar_1^*,\Gamma'\rangle \\ \quad \Delta,\Gamma',\delta \vdash cexp \rightarrow tcexp \\ \quad\quad \Delta,\varnothing \vdash \langle var_2^*,\Gamma'\rangle \rightarrow \langle tvar_2^*,\Gamma''\rangle \\ \quad\quad\quad \Delta,\Gamma'',\delta,\Downarrow typeOf(tcexp) \vdash inst \rightarrow tinst\end{array}}{\begin{array}{l}\Delta,\delta \vdash \langle \texttt{method}\ id\ var_1^*\ cexp\ var_2^*\ inst,\Gamma\rangle \rightarrow \\ \quad \langle(\texttt{method}\ id\ tvar_1^*\ tcexp\ tvar_2^*\ tinst) : formalTypes(tvar_1^*) \rightarrow\Downarrow typeOf(tcexp), \\ \quad\quad \Gamma[formalTypes(tvar_1^*) \rightarrow\Downarrow typeOf(tcexp)/\delta@id]\rangle\end{array}} \quad (8)$$

$$inheritedMethod(\delta@id,\Delta,\Gamma) \Longrightarrow$$
$$contravariant(inhMethType(id,\Delta,\Gamma),formalTypes(tvar_1^*)\rightarrow\Downarrow typeOf(tcexp),\Delta)$$

Vérification des instructions I

- Suivant ce qui a été écrit précédemment, la vérification de types des instructions se fait dans le contexte de la relation d'héritage, de l'environnement de typage, de la classe courante et du type de retour courant. On définit donc la relation :

$$\Delta, \Gamma, \delta, \tau \vdash \mathit{inst} \rightarrow \mathit{tinst}$$

$$\frac{\Delta, \Gamma, \delta, \tau \vdash \mathit{inst}_1 \rightarrow \mathit{tinst}_1 \qquad \Delta, \Gamma, \delta, \tau \vdash \mathit{inst}_2 \rightarrow \mathit{tinst}_2}{\Delta, \Gamma, \delta, \tau \vdash \texttt{seq}\ \mathit{inst}_1\ \mathit{inst}_2 \rightarrow (\texttt{seq}\ \mathit{tinst}_1\ \mathit{tinst}_2) : \texttt{id Void}} \quad (9)$$

$$\frac{\begin{array}{c}\Delta, \Gamma, \delta \vdash \mathit{id} \rightarrow \mathit{id} : T \\ \Delta, \Gamma, \delta \vdash \mathit{exp} \rightarrow \mathit{texp}\end{array}}{\begin{array}{l}\Delta, \Gamma, \delta, \tau \vdash \texttt{assign}\ \mathit{id}\ \mathit{exp} \rightarrow \\ \quad (\texttt{assign}\ \mathit{id} : T\ \mathit{texp}) : \texttt{id Void}\end{array}} \quad \mathit{conformsTo}(\mathit{typeOf}(\mathit{texp}), T) \quad (10)$$

Vérification des instructions II

$$\frac{\Delta,\Gamma,\delta \vdash \mathit{exp} \rightarrow \mathit{texp}}{\Delta,\Gamma,\delta,\tau \vdash \texttt{return}\ \mathit{exp} \rightarrow (\texttt{return}\ \mathit{texp}) : \texttt{id Void}} \quad \mathit{conformsTo}(\mathit{typeOf}(\mathit{texp}),\tau) \tag{11}$$

$$\frac{\Delta,\Gamma,\delta \vdash \mathit{exp} \rightarrow \mathit{texp}}{\Delta,\Gamma,\delta,\tau \vdash \texttt{writeln}\ \mathit{exp} \rightarrow (\texttt{writeln}\ \mathit{texp}) : \texttt{id Void}} \tag{12}$$

$$\frac{\Delta,\Gamma,\delta \vdash \mathit{exp}_1 \rightarrow \mathit{texp}_1 \quad \Delta,\Gamma,\delta \vdash \mathit{exp}_2 \rightarrow \mathit{texp}_2}{\Delta,\Gamma,\delta,\tau \vdash \texttt{writefield}\ \mathit{exp}_1\ \mathit{id}\ \mathit{exp}_2 \rightarrow (\texttt{writefield}\ \mathit{texp}_1\ \mathit{id}\ \mathit{texp}_2) : \texttt{id Void}} \quad \begin{array}{l}\mathit{conformsTo}(\mathit{typeOf}(\mathit{texp}_2),\\ \mathit{fieldType}(\mathit{typeOf}(\mathit{texp}_1)@\mathit{id},\Delta,\Gamma))\end{array} \tag{13}$$

Vérification des instructions III

$$\frac{\Delta,\Gamma,\delta \vdash \mathit{exp} \rightarrow \mathit{texp} \quad \Delta,\Gamma,\delta,\tau \vdash \mathit{inst}_1 \rightarrow \mathit{tinst}_1 \quad \Delta,\Gamma,\delta,\tau \vdash \mathit{inst}_2 \rightarrow \mathit{tinst}_2}{\Delta,\Gamma,\delta,\tau \vdash \texttt{if}\ \mathit{exp}\ \mathit{inst}_1\ \mathit{inst}_2 \rightarrow (\texttt{if}\ \mathit{texp}\ \mathit{tinst}_1\ \mathit{tinst}_2) : \texttt{id Void}}\ \mathit{typeOf}(\mathit{texp}) = \texttt{id Bool} \tag{14}$$

$$\frac{\Delta,\Gamma,\delta \vdash \mathit{exp} \rightarrow \mathit{texp} \quad \Delta,\Gamma,\delta,\tau \vdash \mathit{inst} \rightarrow \mathit{tinst}}{\Delta,\Gamma,\delta,\tau \vdash \texttt{while}\ \mathit{exp}\ \mathit{inst} \rightarrow (\texttt{while}\ \mathit{texp}\ \mathit{tinst}) : \texttt{id Void}}\ \mathit{typeOf}(\mathit{texp}) = \texttt{id Bool} \tag{15}$$

Vérification des types des expressions

- Suivant également ce qui a été écrit précédemment, la vérification de types des expressions se fait dans le contexte de la relation d'héritage, de l'environnement de typage et de la classe courante. On définit donc la relation :

$$\Delta, \Gamma, \delta \vdash \mathit{exp} \rightarrow \mathit{texp}$$

Vérification des types des expressions arithmétiques

$$\Delta,\Gamma,\delta \vdash \texttt{int}\ n \rightarrow (\texttt{int}\ n) : \texttt{id Int} \tag{16}$$

$$\frac{\Delta,\Gamma,\delta \vdash exp_1 \rightarrow texp_1 \qquad \Delta,\Gamma,\delta \vdash exp_2 \rightarrow texp_2}{\Delta,\Gamma,\delta \vdash \texttt{plus}\ exp_1\ exp_2 \rightarrow (\texttt{plus}\ texp_1\ texp_2) : \texttt{id Int}} \quad typeOf(texp_1)=typeOf(texp_2)=\texttt{id Int} \tag{17}$$

$$\frac{\Delta,\Gamma,\delta \vdash exp_1 \rightarrow texp_1 \qquad \Delta,\Gamma,\delta \vdash exp_2 \rightarrow texp_2}{\Delta,\Gamma,\delta \vdash \texttt{minus}\ exp_1\ exp_2 \rightarrow (\texttt{minus}\ texp_1\ texp_2) : \texttt{id Int}} \quad typeOf(texp_1)=typeOf(texp_2)=\texttt{id Int} \tag{18}$$

$$\frac{\Delta,\Gamma,\delta \vdash exp_1 \rightarrow texp_1 \qquad \Delta,\Gamma,\delta \vdash exp_2 \rightarrow texp_2}{\Delta,\Gamma,\delta \vdash \texttt{times}\ exp_1\ exp_2 \rightarrow (\texttt{times}\ texp_1\ texp_2) : \texttt{id Int}} \quad typeOf(texp_1)=typeOf(texp_2)=\texttt{id Int} \tag{19}$$

Vérification des types des expressions booléennes I

$$\Delta, \Gamma, \delta \vdash \texttt{true} \rightarrow \texttt{true} : \texttt{id Bool} \tag{20}$$

$$\Delta, \Gamma, \delta \vdash \texttt{false} \rightarrow \texttt{false} : \texttt{id Bool} \tag{21}$$

$$\frac{\Delta, \Gamma, \delta \vdash \mathit{exp}_1 \rightarrow \mathit{texp}_1 \quad \Delta, \Gamma, \delta \vdash \mathit{exp}_2 \rightarrow \mathit{texp}_2}{\Delta, \Gamma, \delta \vdash \texttt{or}\ \mathit{exp}_1\ \mathit{exp}_2 \rightarrow (\texttt{or}\ \mathit{texp}_1\ \mathit{texp}_2) : \texttt{id Bool}} \quad \mathit{typeOf}(\mathit{texp}_1) = \mathit{typeOf}(\mathit{texp}_2) = \texttt{id Bool} \tag{22}$$

$$\frac{\Delta, \Gamma, \delta \vdash \mathit{exp}_1 \rightarrow \mathit{texp}_1 \quad \Delta, \Gamma, \delta \vdash \mathit{exp}_2 \rightarrow \mathit{texp}_2}{\Delta, \Gamma, \delta \vdash \texttt{and}\ \mathit{exp}_1\ \mathit{exp}_2 \rightarrow (\texttt{and}\ \mathit{texp}_1\ \mathit{texp}_2) : \texttt{id Bool}} \quad \mathit{typeOf}(\mathit{texp}_1) = \mathit{typeOf}(\mathit{texp}_2) = \texttt{id Bool} \tag{23}$$

Vérification des types des expressions booléennes II

$$\frac{\Delta,\Gamma,\delta \vdash exp_1 \rightarrow texp_1 \qquad \Delta,\Gamma,\delta \vdash exp_2 \rightarrow texp_2}{\Delta,\Gamma,\delta \vdash \texttt{less}\ exp_1\ exp_2 \rightarrow (\texttt{less}\ texp_1\ texp_2) : \texttt{id Bool}} \quad typeOf(texp_1) = typeOf(texp_2) = \texttt{id Int} \tag{24}$$

$$\frac{\Delta,\Gamma,\delta \vdash exp_1 \rightarrow texp_1 \qquad \Delta,\Gamma,\delta \vdash exp_2 \rightarrow texp_2}{\Delta,\Gamma,\delta \vdash \texttt{equal}\ exp_1\ exp_2 \rightarrow (\texttt{equal}\ texp_1\ texp_2) : \texttt{id Bool}} \quad \begin{array}{l} conformsTo(typeOf(texp_1), typeOf(texp_2)) \\ \vee\ conformsTo(typeOf(texp_2), typeOf(texp_1)) \end{array} \tag{25}$$

Vérification des types des expressions booléennes III

$$\frac{\Delta,\Gamma,\delta \vdash \mathit{exp} \rightarrow \mathit{texp}}{\Delta,\Gamma,\delta \vdash \texttt{not}\ \mathit{exp} \rightarrow (\texttt{not}\ \mathit{texp}) : \texttt{id Bool}}\ \mathit{typeOf}(\mathit{texp}) = \texttt{id Bool} \tag{26}$$

$$\frac{\Delta,\Gamma,\delta \vdash \mathit{exp} \rightarrow \mathit{texp} \qquad \Delta,\Gamma,\delta \vdash \mathit{cexp} \rightarrow \mathit{tcexp}}{\Delta,\Gamma,\delta \vdash \texttt{instanceOf}\ \mathit{exp}\ \mathit{cexp} \rightarrow (\texttt{instanceOf}\ \mathit{texp}\ \mathit{tcexp}) : \texttt{id Bool}} \tag{27}$$

Vérification des types des expressions sur les objets I

$$\Delta,\Gamma,\delta \vdash \texttt{cexp}\ id \rightarrow (\texttt{cexp}\ id) : aType(id) \qquad isType(id,\Delta) \tag{28}$$

$$\Delta,\Gamma,\delta \vdash \texttt{nil} \rightarrow \texttt{nil} : \texttt{id Object} \tag{29}$$

$$\Delta,\Gamma,\delta \vdash \texttt{self} \rightarrow \texttt{self} : \delta \tag{30}$$

$$\Delta,\Gamma,\delta \vdash \texttt{super} \rightarrow \texttt{super} : \Delta(\delta) \tag{31}$$

$$\frac{\Delta,\Gamma,\delta \vdash cexp \rightarrow tcexp}{\Delta,\Gamma,\delta \vdash \texttt{new}\ cexp \rightarrow (\texttt{new}\ tcexp) :\Downarrow typeOf(tcexp)} \tag{32}$$

$$\frac{\Delta,\Gamma,\delta \vdash exp \rightarrow texp}{\begin{array}{l}\Delta,\Gamma,\delta \vdash \texttt{readField}\ exp\ id \rightarrow \\ \qquad (\texttt{readField}\ texp\ id) : fieldType(typeOf(texp)@id,\Delta,\Gamma)\end{array}} \tag{33}$$

Vérification des types des expressions sur les objets II

$$\frac{\begin{array}{l}\Delta,\Gamma,\delta \vdash \mathit{exp} \rightarrow \mathit{texp} \\ \quad \Delta,\Gamma,\delta \vdash \mathit{exp}^* \rightarrow \mathit{texp}^* \\ \quad\quad TF \rightarrow RT = \\ \quad\quad\quad \mathit{methodType}(\mathit{typeOf}(\mathit{texp})@\mathit{id},\Delta,\Gamma)\end{array}}{\begin{array}{l}\Delta,\Gamma,\delta \vdash \texttt{methodCall}\ \mathit{exp}\ \mathit{id}\ \mathit{exp}^* \rightarrow \\ \quad (\texttt{methodCall}\ \mathit{texp}\ \mathit{id}\ \mathit{texp}^*) : RT\end{array}}\ \mathit{actualsConformTo}(\mathit{texp}^*, TF, \Delta) \tag{34}$$

$$\Delta,\Gamma,\delta \vdash \texttt{id} \rightarrow \texttt{id} : \Gamma(\texttt{id}) \tag{35}$$

Fonctions auxiliaires

typeOf : retourne le type associé à un AST typé.

formalTypes : extrait les types d'une liste de variables typées pour les retourner sous la forme $T_1 \times ... \times T_n$.

inheritedMethod : vrai s'il existe une méthode héritée du nom demandé à partir de la classe donnée.

inhMethodType : retourne le type de la méthode héritée, s'il y en a une.

contravariant : vrai si les types fonctionnels sont contravariants.

conformsTo : vrai si les types sont conformes.

actualsConformTo : vrai si les types des paramètres réels sont conformes aux types des paramètres formels d'une méthode.

fieldType : retourne le type du premier champ du nom donné déclaré ou hérité à partir de la classe donnée.

Activités complémentaires

- Regarder le document de Plotkin sur les SOS.
- Lire les programmes Prolog implantant les analyseurs et le vérificateur de type pour BOPL.

On commence le 3e cours aujourdhui

Cours 3

Introduction aux sémantiques opérationnelles structurelles

Concepts centraux

- 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.
- 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.
- 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.

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

- Une bonne machine virtuelle réalise un compromis entre finesse de la description et complexité des sémantiques résultantes.
  - 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

- 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.
- 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.
- 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.

« *Big step* » et « *Small step* » II

- 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,
  - 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.