MI030 — APS
Analyse des programmes et sémantique

© Jacques Malenfant, 2010–2014

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

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

Cours 2

Analyses statiques et vérification des types

Propriétés dépendantes du contexte I

- 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.*

Propriétés dépendantes du contexte II

- 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).

Principes des analyses statiques I

- Les analyses statiques servent à inférer et vérifier des propriétés sur les programmes.
- Parmi les analyses statiques, celles qui sont dîtes *dirigées par la syntaxe* s'appliquent sur les arbres de syntaxe abstraite.
- Il s'agit de définir des *règles d'inférence* qui vont permettre d'inférer et de vérifier progressivement les propriétés d'un programme en partant des constructions élémentaires puis, de constructions composées en constructions composées, arriver jusqu'au programme en entier.

Principes des analyses statiques II

- Autrement dit, il s'agit d'inférer et vérifier en parcourant l'arbre de syntaxe abstraite de manière complète et systématique.

Définition des analyses statiques I

- 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 des analyses statiques II

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 :

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

Déduction naturelle I

- 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 I

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.

Vérification des types dans les expressions arithmétiques II

- 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).

Les expressions arithmétiques simples I

- 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

Les expressions arithmétiques simples II

programme annoté, nous utilisons une nouvelle grammaire abstraite :

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

- 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.

Règles d'inférences pour le typage I

- 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 de la forme :

$$e \rightarrow te$$

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

Règles d'inférences pour le typage II

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

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

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

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

Typage d'une expression I

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

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

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 :

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

  La grammaire abstraite annotée :

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

Introduction des variables II

- 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.
- Récupérer le type de la variable peut se faire par un environnement de typage. Notons $\Gamma : id \rightarrow t$ 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 l'environnement de typage de la forme :

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

Introduction des variables III

- Pour les règles précédentes cela donne :

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

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

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

Introduction des variables IV

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

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

- 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.

Le système de types de BOPL I

- 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 types prédéfinis sont désignés par des identifiants BOPL, de la même manière que les classes, pour des raisons d'uniformité.

Le système de types de BOPL II

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

$$FormalType_1 \times ... \times FormalType_n \rightarrow ReturnType$$

Les déclarations de classes I

- 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.

Les déclarations de classes II

- Pour conserver ces informations, nous décidons de définir deux fonctions :

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

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

Le corps du programme I

- 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 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`,

Le corps du programme II

- 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 I

- 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.

tprogram ::= `program` *tclass** *tvar** *tinst* : *type*

tclass ::= `class` *id tcexp tvar** *tmethod** : *type*

Grammaire abstraite *typée* de BOPL II

tcexp ::= `cexp` *id* : *type*

tmethod ::= `method` *id tvar** *id tvar** *tseq* : *type*

tvar ::= `var` *tcexp id* : *type*

tinst ::= `seq` *tinst tinst* : *type* | `assign` *id texp* : *type* |
`writefield` *texp id texp* : *type* |
`if` *texp tinst tinst* : *type* | `while` *texp tinst* : *type* |
`return` *texp* : *type* | `writeln` *texp* : *type*

Grammaire abstraite *typée* de BOPL III

$$
\begin{array}{rcl}
\mathit{texp} & ::= & \mathit{int} : \mathit{type} \mid \texttt{true} : \mathit{type} \mid \texttt{false} : \mathit{type} \mid \texttt{not}\ \mathit{texp} : \mathit{type} \mid \\
& & \texttt{nil} : \mathit{type} \mid \texttt{self} : \mathit{type} \mid \texttt{super} : \mathit{type} \mid \\
& & \texttt{new}\ \mathit{tcexp} : \mathit{type} \mid \texttt{instanceof}\ \mathit{exp}\ \mathit{exp} : \mathit{type} \mid \\
& & \mathit{id} : \mathit{type} \mid \texttt{methodcall}\ \mathit{exp}\ \mathit{id}\ \mathit{exp}^* : \mathit{type} \mid \\
& & \texttt{readfield}\ \mathit{exp}\ \mathit{id} : \mathit{type} \mid \texttt{plus}\ \mathit{exp}\ \mathit{exp} : \mathit{type} \mid \\
& & \texttt{minus}\ \mathit{exp}\ \mathit{exp} : \mathit{type} \mid \texttt{times}\ \mathit{exp}\ \mathit{exp} : \mathit{type} \mid \\
& & \texttt{equal}\ \mathit{exp}\ \mathit{exp} : \mathit{type} \mid \texttt{and}\ \mathit{exp}\ \mathit{exp} : \mathit{type} \mid \\
& & \texttt{or}\ \mathit{exp}\ \mathit{exp} : \mathit{type} \mid \texttt{less}\ \mathit{exp}\ \mathit{exp} : \mathit{type} \\
\mathit{type} & ::= & \mathit{id} \mid \mathit{aType}(\mathit{id}) \mid \texttt{id Bool} \mid \texttt{id Int} \mid \texttt{id Object} \mid \texttt{id Void}
\end{array}
$$

Encore quelques éléments de contexte : méthodes I

- 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.

Encore quelques éléments de contexte : méthodes II

- On introduit donc un nouvel élément de contexte pour vérifier le type des instructions :

  $\tau : \mathit{Id}$ type de retour courant.

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

- 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.

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

- 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.

Vérification du programme I

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

  $$program \rightarrow tprogram$$

- 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.

Vérification du programme II

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

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).

Traitement des déclarations de variables locales II

- 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 III

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

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

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

Vérification des déclarations de classes I

- 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 :

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

Vérification des déclarations de classes II

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

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

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.

Vérification de chaque déclaration de classe II

- Les déclarations de méthodes introduisent les types de ces dernières dans l'environnement de typage.

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

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

Vérification des déclarations de méthodes I

- 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 :

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

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

Vérification des déclarations de méthodes II

- 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 I

- 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.

Vérification de chaque méthode II

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

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

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 :

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

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

Vérification des instructions II

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

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

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

Vérification des instructions III

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

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

Vérification des instructions IV

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

Vérification des types des expressions I

- 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 I

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

$$\frac{\Delta,\Gamma,\delta \vdash \mathit{exp}_1 \rightarrow \mathit{texp}_1 \qquad \Delta,\Gamma,\delta \vdash \mathit{exp}_2 \rightarrow \mathit{texp}_2}{\Delta,\Gamma,\delta \vdash \texttt{plus}\ \mathit{exp}_1\ \mathit{exp}_2 \rightarrow (\texttt{plus}\ \mathit{texp}_1\ \mathit{texp}_2) : \texttt{id Int}}\ \mathit{typeOf}(\mathit{texp}_1) = \mathit{typeOf}(\mathit{texp}_2) = \texttt{id Int} \quad (17)$$

$$\frac{\Delta,\Gamma,\delta \vdash \mathit{exp}_1 \rightarrow \mathit{texp}_1 \qquad \Delta,\Gamma,\delta \vdash \mathit{exp}_2 \rightarrow \mathit{texp}_2}{\Delta,\Gamma,\delta \vdash \texttt{minus}\ \mathit{exp}_1\ \mathit{exp}_2 \rightarrow (\texttt{minus}\ \mathit{texp}_1\ \mathit{texp}_2) : \texttt{id Int}}\ \mathit{typeOf}(\mathit{texp}_1) = \mathit{typeOf}(\mathit{texp}_2) = \texttt{id Int} \quad (18)$$

Vérification des types des expressions arithmétiques II

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

Vérification des types des expressions booléennes I

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

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

$$\frac{\Delta,\Gamma,\delta \vdash exp_1 \rightarrow texp_1 \quad \Delta,\Gamma,\delta \vdash exp_2 \rightarrow texp_2}{\Delta,\Gamma,\delta \vdash \texttt{or}\ exp_1\ exp_2 \rightarrow (\texttt{or}\ texp_1\ texp_2) : \texttt{id Bool}} \; typeOf(texp_1)=typeOf(texp_2)=\texttt{id Bool} \quad (22)$$

Vérification des types des expressions booléennes II

$$\frac{\Delta,\Gamma,\delta \vdash exp_1 \rightarrow texp_1 \quad \Delta,\Gamma,\delta \vdash exp_2 \rightarrow texp_2}{\Delta,\Gamma,\delta \vdash \texttt{and}\ exp_1\ exp_2 \rightarrow (\texttt{and}\ texp_1\ texp_2) : \texttt{id Bool}} \; typeOf(texp_1)=typeOf(texp_2)=\texttt{id Bool} \quad (23)$$

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

Vérification des types des expressions booléennes III

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

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

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

Vérification des types des expressions sur les objets I

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

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

$$\Delta,\Gamma,\delta \vdash \texttt{id}\ \texttt{self} \rightarrow \texttt{id}\ \texttt{self} : \delta \qquad (30)$$

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

Vérification des types des expressions sur les objets II

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

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

Vérification des types des expressions sur les objets III

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

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

Fonctions auxiliaires I

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.

Fonctions auxiliaires II

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 I

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