IFIPS – Cycle Ingénieur 2e année Informatique
Projet de compilation 2006–2007

Compilateur J--

deuxième partie : analyse sémantique

1 Analyse sémantique

Cette deuxième partie consiste à vérifier la conformité du fichier source, une fois l'analyse syntaxique effectuée avec succès. Dans tout ce qui suit, les types sont de la forme suivante :

$$\tau ::= \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{string} \mid C \mid \texttt{typenull}$$

où C désigne une classe.

1.1 Héritage et sous-typage

On notera $C_1 \longrightarrow C_2$ la relation « la classe C_1 est une sous-classe de la classe C_2 », qui est la clôture réflexive-transitive de l'ensemble des déclarations d'héritage C_1 `inherits` C_2 contenue dans le source.

Il y a une classe prédéfinie, *Object*, qui n'hérite d'aucune classe. Toute classe qui n'est pas déclarée comme héritant d'une autre hérite de la classe *Object*.

La relation de sous-typage $\tau_1 \sqsubseteq \tau_2$ signifie « le type τ_1 est un sous-type du type τ_2 » et est définie ainsi :

- si τ est `bool`, `int`, `string` ou `unit`, alors $\tau \sqsubseteq \tau$;
- si $C_1 \longrightarrow C_2$ alors $C_1 \sqsubseteq C_2$;
- enfin, `typenull` $\sqsubseteq C$.

Intuitivement, on peut interpréter la relation $\tau_1 \sqsubseteq \tau_2$ par « toute valeur de type τ_1 peut être donnée là où est attendue une valeur de type τ_2 ». On dit que les types τ_1 et τ_2 sont *compatibles*, et on note $\tau_1 \equiv \tau_2$, si $\tau_1 \sqsubseteq \tau_2$ ou $\tau_2 \sqsubseteq \tau_1$; dans ce cas, on note $\tau_1 \sqcup \tau_2$ le plus grand de ces deux types.

La notion de sous-typage est étendue aux profils (listes de types) de la manière suivante :

$$(\tau_1, \ldots, \tau_n) \sqsubseteq (\tau'_1, \ldots, \tau'_n) \text{ si et seulement si } \tau_i \sqsubseteq \tau'_i \text{ pour tout } i \in 1, \ldots n.$$

1.2 Attributs, constructeurs et méthodes d'une classe

Soit C une classe. On dit que la classe C possède un attribut x de type τ si elle-même ou l'une de ses sur-classes déclare un attribut x de type τ.

De même, on dit que la classe C possède une méthode $\tau\ m(\tau_1, \ldots, \tau_n)$ si elle-même ou l'une de ses sur-classes déclare une méthode de tel profil.

En revanche, on dit que la classe C possède un constructeur $C(\tau_1, \ldots, \tau_n)$ si elle déclare *elle-même* un constructeur de tel profil. On suppose un unique constructeur dans la classe *Object* ne prenant aucun argument.

1.3 Primitives

Le langage comporte un ensemble de primitives, qui peuvent être vues comme des méthodes statiques prédéfinies. La liste de ces primitives avec leur profil est la suivante :

```
unit open_screen(int, int)
unit clear_screen()
unit set_color(int, int, int)
unit draw_point(int, int)
unit draw_line(int, int, int, int)
unit draw_circle(int, int, int)
unit draw_rect(int, int, int, int)
unit fill_rect(int, int, int, int)
unit usleep(int)
int get()
unit set(int)
bool move()
unit turn_left()
unit turn_right()
```

De plus, il existe une primitive particulière, `print`, acceptant un nombre quelconque d'arguments et dont la valeur de retour est de type `unit`.

1.4 Typage des expressions

Dans toute la suite, C_0 désigne la classe courante, c'est-à-dire la classe pour laquelle on est en train de réaliser l'analyse sémantique.

Un contexte Γ est un ensemble de déclarations de variables noté $\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n$. Il ne sera utilisé que pour les variables locales, les arguments de constructeurs et de méthodes. Un contexte Γ étant donné, une expression e est bien typée de type τ si et seulement si l'une des conditions suivantes est satisfaite :

- e est une constante de type τ ;
- e est `null` et $\tau =$ `typenull` ;
- e est `nop` et $\tau =$ `unit` ;
- e est `this` et $\tau = C_0$;
- e est l'identificateur x et il y a au moins une déclaration pour x dans Γ dont la plus récente est $\tau\ x$;
- e est l'identificateur x, il n'y a pas de déclaration pour x dans Γ et la classe C_0 possède un attribut x de type τ ;
- e est $e_1.x$, l'expression e_1 a le type C et C possède un attribut x de type τ ;
- e est e_1:=e_2, l'expression e_1 a le type τ, l'expression e_2 a le type τ_2 et $\tau_2 \sqsubseteq \tau$;
- e est -e_1, l'expression e_1 a le type `int` et $\tau =$ `int` ;
- e est `not` e_1, l'expression e_1 a le type `bool` et $\tau =$ `bool` ;
- e est e_1=e_2 ou e_1<>e_2, e_1 a le type τ_1, e_2 a le type τ_2, les types τ_1 et τ_2 sont compatibles et $\tau =$ `bool` ;
- e est e_1<e_2, e_1<=e_2, e_1>e_2, ou e_1>=e_2, e_1 et e_2 ont le type `int` et $\tau =$ `bool` ;
- e est e_1+e_2, e_1-e_2, e_1*e_2, ou e_1/e_2, e_1 et e_2 ont le type `int` et $\tau =$ `int` ;

- e est e_1 `and` e_2 ou e_1 `or` e_2, e_1 et e_2 ont le type `bool` et $\tau =$ `bool` ;
- e est $m(e_1, \dots, e_n)$, chaque e_i est de type τ_i et la classe C_0 possède une unique méthode $\tau\ m(\tau'_1, \dots, \tau'_n)$ telle que $(\tau_1, \dots, \tau_n) \sqsubseteq (\tau'_1, \dots, \tau'_n)$ (attention : cela ne veut pas dire que l'on ne peut pas redéfinir une méthode, car dans le cas d'une redéfinition on a plusieurs fois le *même profil* et donc une seule méthode du point de vue du typage) ;
- e est $p(e_1, \dots, e_n)$, chaque e_i est de type τ_i et p est une primitive de profil $\tau\ p(\tau'_1, \dots, \tau'_n)$ avec $(\tau_1, \dots, \tau_n) \sqsubseteq (\tau'_1, \dots, \tau'_n)$;
- e est `print`$(e_1, \dots, e_n)$ avec chaque e_i de type `int`, `bool` ou `string` et $\tau =$ `unit` ;
- e est $e'.m(e_1, \dots, e_n)$, avec e' de type C, chaque e_i est de type τ_i et la classe C possède une unique méthode $\tau\ m(\tau'_1, \dots, \tau'_n)$ telle que $(\tau_1, \dots, \tau_n) \sqsubseteq (\tau'_1, \dots, \tau'_n)$;
- e est `new` $C(e_1, \dots, e_n)$, chaque e_i est de type τ_i et la classe C possède un unique construceur $C(\tau'_1, \dots, \tau'_n)$ tel que $(\tau_1, \dots, \tau_n) \sqsubseteq (\tau'_1, \dots, \tau'_n)$;
- e est `let` $x = e_1$ `in` e_2, e_1 est de type τ_1 (dans Γ) et e_2 est de type τ dans le contexte $\Gamma, \tau_1\ x$;
- e est e_1 `;` e_2, e_1 est de type τ_1 et e_2 de type τ ;
- e est `if` e_1 `then` e_2 `else` e_3, e_1 est de type `bool`, e_2 de type τ_2, e_3 de type τ_3, les types τ_2 et τ_3 sont compatibles et $\tau = \tau_2 \sqcup \tau_3$;
- e est `if` e_1 `then` e_2, e_1 est de type `bool`, e_2 de type `unit` et $\tau =$ `unit` ;
- e est `while` e_1 `do` e_2, e_1 est de type `bool`, e_2 de type τ_2 et $\tau =$ `unit`.

1.5 Typage des classes

1.5.1 Conditions d'existence et d'unicité

Pour être sémantiquement correct, un fichier doit tout d'abord respecter les contraintes suivantes :

- chaque classe ne peut être définie qu'une seule fois et ne peut avoir pour nom `Object` ;
- une classe ne peut hériter d'une classe qui n'est pas définie ;
- la relation d'héritage ne peut admettre de cycle.

Notez bien que l'ordre d'apparition des classes dans le fichier n'a aucune importance : en tout point, on peut référencer une classe définie ailleurs dans le fichier (avant ou après).

D'autre part, les conditions suivantes doivent être remplies :

- tous les attributs d'une classe doivent porter des noms différents (cela inclut les attributs définis dans les sur-classes) ;
- tous les constructeurs d'une même classe doivent avoir des profils différents et porter le même nom que la classe ;
- une méthode ne peut pas porter le nom d'une primitive ni de `print` ;
- deux méthodes de même nom définies à l'intérieur d'un même classe doivent avoir des profils différents ;
- deux méthodes de même nom et de même profil définies dans une classe et l'un de ses sous-classes doivent avoir le même type de retour.

1.5.2 Typage des attributs, constructeurs et méthodes

Soit C_0 la classe courante.

Typage des attributs. Pour une déclaration d'attributs $\tau\ x_1, \dots, x_n$ on se contente de vérifier que le type τ est bien formé (c'est-à-dire que la classe existe lorsqu'il s'agit d'une classe).

Typage des constructeurs. Un constructeur $C_0(\tau_1\ x_1; \dots; \tau_n\ x_n)\ e$ est bien formé si tous les identificateurs x_i sont différents deux à deux, tous les types τ_i bien formés, et si l'expression e est bien typée dans le contexte $\tau_1\ x_1, \dots, \tau_n\ x_n$.

Typage des méthodes. Une méthode $\tau\ m(\tau_1\ x_1; \dots; \tau_n\ x_n)\ e$ est bien formée si tous les identificateurs x_i sont différents deux à deux, le type τ et les types τ_i bien formés, si l'expression e est bien typée de type τ' dans le contexte $\tau_1\ x_1, \dots, \tau_n\ x_n$ et enfin si soit $\tau = \texttt{unit}$, soit $\tau' \sqsubseteq \tau$.

1.6 Indications

On suggère fortement de réaliser l'analyse sémantique en trois phases distinctes, à savoir :

1. déclaration des classes et vérification de leur unicité,
2. construction de la relation d'héritage, déclaration des attributs, constructeurs et méthodes de toutes les classes (avec les vérifications adéquates),
3. typage des corps des constructeurs, des méthodes et de l'expression principale.

2 Travail demandé

Adapter votre compilateur pour qu'il effectue la passe d'analyse sémantique, à l'issue de l'analyse syntaxique. En plus du fichier `J--` en argument, votre compilateur devra reconnaître désormais deux options, `-parse-only` et `-type-only`. La première spécifie de s'arrêter juste après l'analyse syntaxique. La seconde est sans effet dans cette deuxième partie et son utilisation sera justifiée par la suite.

En plus des erreurs lexicales et syntaxiques, votre compilateur devra maintenant afficher les erreurs sémantiques et, le cas échéant, terminer avec le code de sortie 1 (`System.exit(1)`). Les erreurs sémantiques seront signalées de la même manière que les erreurs lexicales et syntaxiques ; par exemple :

```
File "test.jmm", line 21, characters 13-14:
cette expression a le type int mais devrait avoir le type string
```

Le comportement en cas d'absence d'erreur ou d'erreur du compilateur ne change pas.

Modalités de remise de votre projet. Votre projet doit se présenter sous forme d'une archive contenant les *sources* de votre programme (ne donnez pas les fichiers compilés) ainsi qu'un court rapport expliquant les difficultés rencontrées et les solutions adoptées. Il doit être envoyé par e-mail à `Jean-Christophe.Filliatre@lri.fr`.