Algorithmique et Approche Fonctionnelle de la Programmation

Cours 2 : Sémantique et typage

21 septembre 2009

Louis Mandel
Université Paris-Sud 11
louis.mandel@lri.fr

Sémantique des langage

Définition formelle des langages de programmation

- définir le comportement des programmes
  - ▷ comprendre ce qu'un programme calcule
  - ▷ raisonner sur les propriétés du programme
- définir des analyses sur langage
- référence pour implanter des compilateurs

Le langage

Nous allons étudier le sous ensemble de OCaml suivant :

$$
\begin{array}{lcl}
\textit{expression} & ::= & \textit{constante} \\
& | & \textit{var} \\
& | & \textit{expression}\ \texttt{+}\ \textit{expression} \\
& | & \textit{expression}\ \texttt{-}\ \textit{expression} \\
& | & \ldots \\
& | & \texttt{if}\ \textit{expression}\ \texttt{then}\ \textit{expression}\ \texttt{else}\ \textit{expression} \\
& | & \texttt{let}\ \textit{var}\ \texttt{=}\ \textit{expression}\ \texttt{in}\ \textit{expression} \\
& | & \texttt{fun}\ \textit{var}\ \texttt{->}\ \textit{expression} \\
& | & \textit{expression}\ \textit{expression}
\end{array}
$$

Sémantique à petits pas

- Un programme OCaml peut se traduire en une seule expression en remplaçant les `let` en `let/in`
  - Exemple : l'exécution du programme

    ```
    let f = (fun x -> x / 2);;
    let _ =  f (4 + 4);;
    ```

    est équivalente à l'évaluation de l'expression

    ```
    let f = (fun x -> x / 2) in f (4 + 4)
    ```
- L'évaluation d'une expression est une succession de réductions jusqu'à l'obtension d'une valeur
- Les valeurs sont définies de la manière suivante :

$$\begin{array}{rcl} \textit{valeur} & ::= & \textit{constante} \\ & | & \texttt{fun}\ \textit{var}\ \texttt{->}\ \textit{expression} \end{array}$$

Sémantique à petits pas : exemple

- Pour évaluer l'expression suivante :

  `let f = (fun x -> x / 2) in f (4 + 4)`

  - il faut commencer par `f` par sa définition dans la partie droite du `let`

    $\longrightarrow$ `(fun x -> x / 2) (4 + 4)`
  - puis il faut calculer la valeur de `4 + 4`

    $\longrightarrow$ `(fun x -> x / 2) 8`
  - ensuite, il faut remplacer `x` par 8 dans le corps de la fonction

    $\longrightarrow$ `8 / 2`
  - enfin, il faut calculer le résultat de `8 / 2`

    $\longrightarrow$ `4`

Règles de réductions

- L'évaluation se fait en suivant la structure de l'expression
- Application $e_1\ e_2$:
  - si l'expression de gauche (e_1) est une fonction : `(fun` x `->` e`)` v
  - si l'expression de droite (e_2) est une valeur : v
  - l'expression $e_1\ e_2$ se réduit en l'expression e où la variable x a été substituée (remplacée) par v (on note la substitution $e[x \leftarrow v]$)
  - la règle de l'application est :

$$\texttt{(fun}\ x\ \texttt{->}\ e\texttt{)}\ v \qquad \longrightarrow \qquad e[x \leftarrow v]$$

  - si e_1 ou e_2 ne sont pas des valeurs, on les réduit jusqu'à l'obtention de valeurs

Règles de réductions

- Déclaration locale `let` x `=` e_1 `in` e_2 :
  - si l'expression e_1 est un valeur : v
  - l'expression `let` x `=` v `in` e_2 se réduit en l'expression e_2 où x a été substitué par v
  - la règle de la déclaration locale est :

$$\texttt{let } x \texttt{ = } v \texttt{ in } e_2 \quad \longrightarrow \quad e_2[x \leftarrow v]$$

  - si l'expression e_1 n'est pas une valeur, on la réduit jusqu'à l'obtention d'une valeur

Règles de réductions

- Conditionnelle `if` e `then` e_1 `else` e_2 :
  - si l'expression e_1 est un valeur : `true` ou `false`
  - l'expression `if` e `then` e_1 `else` e_2 se réduit en l'expression e_1 ou e_2 selon la valeur de e
  - les règles de la conditionnelle sont :

$$\texttt{if true then } e_1 \texttt{ else } e_2 \longrightarrow e_1$$

$$\texttt{if false then } e_1 \texttt{ else } e_2 \longrightarrow e_2$$

  - si l'expression e n'est pas une valeur, on la réduit jusqu'à l'obtention d'une valeur

Règles de réductions

- Addition e_1 + e_2 :
  - si l'expression e_1 est une constante entière : n_1
  - si l'expression e_2 est une constante entière : n_2
  - l'expression e_1 + e_2 se réduit en la valeur n si n est le résultat de l'addition entière de n_1 et n_2
  - la règle de l'addition est :

$$n_1 \text{ + } n_2 \longrightarrow n \quad \text{avec} \quad n = n_1 + n_2$$

  - si e_1 ou e_2 ne sont pas des valeurs, on les réduit jusqu'à l'obtention de valeurs

- Les règles de réductions des autres opérateurs sont similaires

Substitution

- Substituer une variable x par une valeur v dans un expression e
  - ▷ est noté $e[x \leftarrow v]$
  - ▷ veut dire « remplacer » x par v dans e
  - ▷ se définit par récurrence sur la structure de e
- si e est une constante c :

  $c[x \leftarrow v] \quad = \quad c$
- si e est une variable y :

  $y[x \leftarrow v] \quad = \quad y$ si $x \neq y$

  $y[x \leftarrow v] \quad = \quad v$ si $x = y$
- si e est une addition e_1 `+` e_2 :

  $(e_1$ `+` $e_2)[x \leftarrow v] \quad = \quad e_1[x \leftarrow v]$ `+` $e_2[x \leftarrow v]$
- si e est une conditionnelle `if` e_1 `then` e_2 `else` e_3 :

  `(if` e_1 `then` e_2 `else` $e_3)[x \leftarrow v] \quad =$

  `if` $e_1[x \leftarrow v]$ `then` $e_2[x \leftarrow v]$ `else` $e_3[x \leftarrow v]$

Substitution (suite)

- si e est une déclaration locale `let` y `=` e_1 `in` e_2 :

  si $x \neq y$

  $$(\texttt{let}\ y\ \texttt{=}\ e_1\ \texttt{in}\ e_2)[x \leftarrow v] \quad = \quad \texttt{let}\ y\ \texttt{=}\ e_1[x \leftarrow v]\ \texttt{in}\ e_2[x \leftarrow v]$$

  si $x = y$

  $$(\texttt{let}\ y\ \texttt{=}\ e_1\ \texttt{in}\ e_2)[x \leftarrow v] \quad = \quad \texttt{let}\ y\ \texttt{=}\ e_1[x \leftarrow v]\ \texttt{in}\ e_2$$

- si e est une fonction `fun` y `->` e' :

  si $x \neq y$

  $$(\texttt{fun}\ y\ \texttt{->}\ e')[x \leftarrow v] \quad = \quad (\texttt{fun}\ y\ \texttt{->}\ e_1[x \leftarrow v])$$

  si $x = y$

  $$(\texttt{fun}\ y\ \texttt{->}\ e')[x \leftarrow v] \quad = \quad (\texttt{fun}\ y\ \texttt{->}\ e_1)$$

- si e est une application $e_1\ \ e_2$:

  $$(e_1\ \ e_2)[x \leftarrow v] \quad = \quad e_1[x \leftarrow v]\ \ e_2[x \leftarrow v]$$

Typage

- Le but du typage est de vérifier à la compilation qu'il ne va pas y avoir d'erreurs lors de l'exécution
- En OCaml, il n'est pas nécessaire pour le programmeur de donner les types
- Le compilateur infère (trouve) les types automatiquement
- L'algorithme de typage fonctionne par récurrence sur la structure des expressions
- Le typage d'un expression se fait dans un environnement H
  - ▷ H associe un type aux variables (libres) de l'expression à typer

- « Dans l'environnement H, l'expression e a le type t » sera noté :

$$H \vdash e : t$$

Règles de typage

- Addition e_1 `+` e_2 :
  - si l'expression e_1 est de type `int`
  - si l'expression e_2 est de type `int`
  - alors e_1 `+` e_2 est de type `int`
  - la règle de l'addition est :

$$\frac{\text{si} \quad H \vdash e_1 : \texttt{int} \quad \text{et} \quad H \vdash e_2 : \texttt{int}}{\text{alors} \quad H \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

  - sinon, l'expression est mal typée

Règles de typage

- Conditionnelle `if` e `then` e_1 `else` e_2 :
  - si l'expression e est de type `bool`
  - si l'expression e_1 est de type t
  - si l'expression e_2 est du même type t
  - alors `if` e `then` e_1 `else` e_2 est de type t
  - la règle de la conditionnelle est :

$$\frac{\text{si} \quad H \vdash e : \texttt{bool} \quad \text{et} \quad H \vdash e_1 : t \quad \text{et} \quad H \vdash e_2 : t}{\text{alors} \quad H \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t}$$

  - sinon, l'expression est mal typée

Règles de typage

- Déclaration locale `let` x `=` e_1 `in` e_2 :
  - si l'expression e_1 est de type t_1
  - si dans l'environnement où x est de type t_1, l'expression e_2 est de type t_2
  - alors `let` x `=` e_1 `in` e_2 est de type t_2
  - la règle de la déclaration locale est :

$$\frac{\text{si} \quad H \vdash e_1 : t_1 \quad \text{et} \quad x : t_1, H \vdash e_2 : t_2}{\text{alors} \quad H \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : t_2}$$

  - sinon, l'expression est mal typée

Règles de typage

- Fonctions `fun` x `->` e :
  - si dans l'environnement où x est de type t_1, l'expression e est de type t_2
  - alors `fun` x `->` e est de type t_1 `->` t_2
  - la règle de typage des fonctions est :

$$\frac{\text{si} \qquad x : t_1, H \vdash e : t_2}{\text{alors} \qquad H \vdash (\texttt{fun}\ x\ \texttt{->}\ e) : t_1\ \texttt{->}\ t_2}$$

  - sinon, l'expression est mal typée

Règles de typage

- Application $e_1\ \ e_2$:
  - si l'expression e_1 est de type t_2 `->` t_1
  - si l'expression e_2 est de type t_2
  - alors $e_1\ \ e_2$ est de type t_1
  - la règle de typage de l'application est :

$$\frac{\text{si} \quad H \vdash e_1 : t_2 \texttt{ -> } t_1 \quad \text{et} \quad H \vdash e_2 : t_2}{\text{alors} \quad H \vdash (e_1\ \ e_2) : t_1}$$

  - sinon, l'expression est mal typée

Règles de typage

- Variable x :
  - ▷ on va regarder dans l'environnement le type associé à la variable
  - ▷ la règle de typage des variables est :

$$\frac{\text{si} \qquad H(x) = t}{\text{alors} \qquad H \vdash x : t}$$

  - ▷ si la variable x n'est pas présente dans l'environnement H, c'est qu'elle n'a pas été définie. Le programme est donc incorrect

Sûreté du typage

Théorème

Si l'expression e est bien typée alors
- soit e se réduit en un nombre fini d'étapes en une valeur
- soit e se réduit à l'infini

Démonstration :

- Préservation du typage par réduction : si l'expression e est bien typée et si e se réduit en e' alors l'expression e' est bien typée et a le même type que e
- Les forme normales sont des valeurs : si l'expression e est bien typée et si e ne peut plus être réduite alors e est une valeur