Analyse sémantique avancée
23 novembre 2011

Introduction

Qualité d'un langage de programmation de haut niveau:

- représenter abstraitement des opérations/données complexes;
- réutiliser le code dans des situations multiples;
- favoriser la compréhension du lecteur.

Le compilateur:

- aider à la détection des erreurs;
- engendrer un code efficace.

Rappels sur les types

- Les types sont des objets structurés qui représentent les ensembles de valeurs possibles pour les résultats des programmes.
- Les types font en général partie du langage de programmation.
- On peut utiliser dans le compilateur des structures de types plus complexes (par exemple des ensembles de type) pour représenter des propriétés du programme.

Arbres de syntaxe abstraite annotés

- Les types peuvent être ajoutés en décoration de l'arbre de syntaxe abstraite.

```
type typ = ....
type profil = ...
type ('a,'b) annot = {annot:'a; asa:'b}
type funcs = (profil,ident) annot
type expr = (typ,expr_raw) annot
and expr_raw =
   | Cst of cte
   | Binop of op * expr * expr
   | If of expr * expr * expr
   | Call of funcs * expr list
...
```

  (Analogue à l'insertion de localisations)

- Le typage prend en argument une expression avec des localisations et produit une expression décorée par des types.

Sommaire

Surcharge

- La surcharge permet d'utiliser le même nom pour des constructions différentes
  - opérations arithmétiques (entiers, flottants)
  - fonctions utilisateurs (Java, ada)
- La portée ne peut être résolue de manière syntaxique.
- Approche sémantique: le contexte d'utilisation d'une expression devra permettre de déterminer sans ambiguité l'opération concernée.

Mise en œuvre

- A l'issue de l'analyse syntaxique, une constante ou une fonction peut représenter des objets calculatoires différents:
  - Une constante numérique peut s'interpréter comme un flottant ou un entier (représentés différemment en machine)
  - Les opérations arithmétiques telles que $+$ peuvent représenter l'addition sur les entiers ou sur les flottants et même des opérations mixtes (en introduisant des coercions).
- A l'issue de l'analyse sémantique on souhaite avoir résolu ces ambiguités : la référence à une opération correspond à un code machine précis.

Résolution statique versus résolution dynamique

- Une fonction machine (code machine) correspond à une suite d'instructions ou bien une étiquette.
- Un même symbole dans le programme source peut correspondre à plusieurs fonctions machines.
- Résolution statique : on détermine à la compilation quelle fonction machine doit être appelée.
- Résolution dynamique : on produit un code unique mais qui fait un choix à l'exécution entre plusieurs fonctions machines.
  - En Java, on accède à l'étiquette du code de la méthode via le descripteur de classe (indirection mais pas de test).
  - En Python, la donnée est stockée avec une étiquette pour son type qui est testée à l'exécution afin de choisir l'instruction à exécuter.

Sommaire

Surcharge à la ADA

- Certaines opérations ont plusieurs profils possibles qui correspondent à des codes machines différents.
- Certaines expressions peuvent avoir plusieurs types.
- La surcharge doit être complètement résolue statiquement : le contexte doit lever les ambiguïtés.
- Analogue au habitudes mathématiques : les mêmes notations représentent des opérations différentes suivant le contexte (addition sur les entiers, les complexes, les matrices . . .), mais le sens est toujours unique.
- Peut aussi servir pour l'optimisation des programmes Python afin de détecter des erreurs et d'engendrer un code plus efficace quand il y a une seule possibilité.

Exemple

Problème

- $+ : ([\texttt{int}; \texttt{int}], \texttt{int})$ et $([\texttt{float}; \texttt{float}], \texttt{float})$
- les constantes telles que $3, 4$ ont les types `int` et `float`
- résoudre $(1 + 4) + 3.2$

Algorithme

- On procède des feuilles vers la racine en collectant l'ensemble des types possibles pour les expressions et les objets

$$\frac{1 : \texttt{int}, \texttt{float} \quad 4 : \texttt{int}, \texttt{float} \quad + : ([\texttt{int}; \texttt{int}], \texttt{int}), ([\texttt{float}; \texttt{float}], \texttt{float})}{(1 + 4) : \texttt{int}, \texttt{float}}$$

$$\frac{(1 + 4) : \texttt{int}, \texttt{float} \quad 3.2 : \texttt{float} \quad + : ([\texttt{float}; \texttt{float}], \texttt{float})}{(1 + 4) + 3.2 : \texttt{float}}$$

Exemple - suite

- S'il y a plus d'un type possible pour l'expression complète, la surcharge ne peut pas être résolue statiquement.
- Sinon il faut s'assurer qu'il y a un seul profil possible pour chaque sous-expression.
- Cela est fait lors d'une phase descendante:

$$\frac{(1+4)+3.2 : \texttt{float}}{(1+4) : \texttt{float} \quad 3.2 : \texttt{float} \quad + : ([\texttt{float};\texttt{float}], \texttt{float})}$$

$$\frac{(1+4) : \texttt{float}}{1 : \texttt{float} \quad 4 : \texttt{float} \quad + : ([\texttt{float};\texttt{float}], \texttt{float})}$$

Sommaire

Typage dans Java

- Dans Java, les types sont soit des types numériques, soit des tableaux, soit des classes.
- Les classes sont organisées suivant une notion d'héritage simple $A \leq B$ si A est une sous-classe de B.
- Si $A \leq B$ alors tout objet de type A est aussi de type B.
  Les types ne sont pas uniques.
- On distingue le type statique déterminé à la compilation du type dynamique à l'exécution.
- Le type statique est une approximation du type dynamique de l'objet:
  - si e a pour type statique A alors toute exécution de e donne un objet dans un type $B \leq A$.
- Le type statique représente un ensemble de types possibles pour une expression (tous les sous-types).

Surcharge statique à la Java

- Utiliser le même nom dans la même classe pour des opérations de profils différents.
- On ne regarde que le type des arguments mais on prend en compte le sous-typage entre classes.
- Une méthode $e.m(e_1, \ldots, e_n)$ avec $e : A$ et $e_i : A_i$. Son profil est $[A; A_1; \cdots A_n]$.
- On cherche toutes les méthodes de noms m définies dans les classes B dont A hérite et qui attendent n arguments. Cela nous donne une collection de profils: $\{...[B; B_1; \cdots B_n]...\}$
- Le profil $[B; B_1; \cdots B_n]$ est adapté à l'appel $e.m(e_1, \ldots, e_n)$ ssi $A \leq B$ et $A_i \leq B_i$.
  Parmi tous les profils adaptés possibles on cherche s'il y en a un plus petit que les autres.

Exemple

- Deux classes $A \leq B$,
  une méthode m_B dans B avec des arguments $B\ x; B\ y$ ($[B; B; B]$)
  deux méthodes m_{AB} et m_{BA} dans A avec des arguments $A\ x; B\ y$ ($[A; A; B]$) et $B\ x; A\ y$ ($[A; B; A]$)
- Soit $e.m(e_1, e_2)$ avec $e : A$
  - Si $e_1, e_2 : B$ $[A; B; B]$ une solution m_B.
  - Si $e_1 : A$ $e_2 : B$, $[A; A; B]$ deux méthodes applicables m_{AB} et m_B et une seule meilleure m_{AB}.
  - Si $e_1 : A$ $e_2 : A$, $[A; A; A]$ trois méthodes applicables m_{AB}, m_{BA} et m_B mais deux incomparables m_{AB} et m_{BA} : échec.

Sommaire

Vérification versus synthèse de types

- Environnement: associe à chaque variable (symbole de fonction) son type (profil)
- Règles de typage: expliquent sous quelle condition une expression est bien typée dans un environnement.
- Vérification:
  - Les variables sont déclarées avec leur type, les fonctions avec leur profil
  - vérifier qu'un expression a un certain type, éventuellement calculer le type d'une expression
- Synthèse: étant donnée une expression qui peut contenir des variables, existe-t-il des types pour les variables et l'expression tels que le résultat soit bien typé ?

Algorithme de typage

L'utilisateur déclare le type des variables et du retour des fonctions.

- Etant donné un environnement et une expression: calcule son type.
- Etant donné un environnement et un programme vérifie qu'il est bien formé.

$$\frac{\rho + (x_i : \tau_i)_{i=1..n} + (f : [\tau_1; \ldots ; \tau_n], \tau) \vdash e : \tau \quad \rho + (f : [\tau_1; \ldots ; \tau_n], \tau) \vdash p \ \textbf{ok}}{\rho \vdash (\texttt{Fun}(f, [x_1 : \tau_1; \ldots ; x_n : \tau_n], \tau, e) :: p) \ \textbf{ok}}$$

Dans certains cas, il faut vérifier l'égalité entre types:

- application d'un opérateur: conditionnelle, fonction définie par l'utilisateur …
- déclaration d'une fonction (adéquation entre le type du corps et le type de retour déclaré)

Synthèse de type

- Ne pas écrire de types dans les déclarations de fonctions:

$$\frac{\rho + (x_i : \tau_i)_{i=1..n} + (f : [\tau_1 ; \ldots ; \tau_n], \tau) \vdash e : \tau \quad \rho + (f : [\tau_1 ; \ldots ; \tau_n], \tau) \vdash p \ \textbf{ok}}{\rho \vdash (\mathtt{Fun}(f, [x_1 ; \ldots ; x_n], e) :: p) \ \textbf{ok}}$$

- Comment deviner τ_i et τ ?
- Idée : utiliser les contraintes d'égalité.
- Technique : introduire des variables de type.

Sommaire

Cadre fonctionnel

- Un langage dans lequel les fonctions sont des expressions comme les autres: **fun** $x \to e$
  - Pas de distinction déclaration de variable/déclaration de fonction
  - Pas de distinction type/signature : type fonctionnel $\tau_1 \to \tau_2$.
- Fonction unaire:
  - `Def f (x,y) = 2 * x < y` devient
    `let f = fun x → fun y → 2 * x < y`
  - La signature `([int;int],bool)` devient
    le type fonctionnel `int` → `int` → `bool`
    associativité droite `int` → (`int` → `bool`).
- Application binaire: `f(4,2)` devient `f 4 2`
  associativité gauche `((f 4) 2)`
- pas de distinction entre expression et instruction

Description du langage - types

- On introduit des opérateurs de type pour les types de bases (entiers, chaînes, `unit` ...) ou les types paramétrés (tableaux, produit, ...) ou encore les types définis par l'utilisateur.
- Chaque opérateur de type a une arité (nombre de types pris en arguments)
  - type constant (`int`,...): arité 0
  - type (tableau, liste,...) : arité 1
  - type produit (`int*bool`) : arité 2

```
type typop = Tbool | Tint | Tfloat | Tunit ...
           | Tprod | Tarr | Tdef of ident | ...
type typ = Tfun of typ * typ | Tconst of typop * typ list
```

Description du langage

```
type primop = Sum | Diff | Prod | Quot | ...
type cte = Int of int | Float of float | Bool of bool ...
    | Oper of primop
and expr =
    Cst of cte
  | Var of ident
  | App of expr * expr
  | Fun of ident * expr
  | Letin of ident * expr * expr
type prg = (ident * expr) list
```

Règles de typage-expressions

Constante

$$\frac{\texttt{type\_cte}(c) = \tau}{\rho \vdash \texttt{Cst}(c) : \tau}$$

Variable

$$\frac{(x : \tau) \in \rho}{\rho \vdash \texttt{Var}(x) : \tau}$$

Application de fonction

$$\frac{\rho \vdash f : \tau' \rightarrow \tau \quad \rho \vdash e : \tau'}{\rho \vdash \texttt{App}(f, e) : \tau}$$

Fonction

$$\frac{\rho + (x : \tau') \vdash e : \tau}{\rho \vdash \texttt{Fun}(x, e) : \tau' \rightarrow \tau}$$

Déclaration locale

$$\frac{\rho \vdash e_1 : \tau_1 \quad \rho + (x : \tau_1) \vdash e_2 : \tau_2}{\rho \vdash \texttt{Letin}(x, e_1, e_2) : \tau_2}$$

Règle de typage-programmes

$$\frac{\rho \vdash e : \tau \quad \rho + (x : \tau) \vdash p\ \textbf{ok}}{\rho \vdash ((x, e) :: p)\ \textbf{ok}}$$

Mise en œuvre de la synthèse

- Ajout de variables de type α, β,... notées `'a,'b,...` en ocaml.

```
type typ = Tfun of typ * typ
         | Tconst of typop * typ list
         | Tvar of ident
```

- Introduction de contraintes $\alpha = \tau$.

Algorithme de typage

Avec une table globale pour les types des symboles

```
let rec type_prog = function
  [] -> ()
| (x,e)::p -> let t = type_expr e
              in add_typ x t; type_prog p

let rec type_expr = function
  Cte c -> type_cte c
| Var x -> find_typ x
| LetIn(x,e1,e2) -> let t1 = type_expr e1 in
                    add_typ x t1; type_expr e2
```

Algorithme de typage : le cas des fonctions

```
| Fun(x,e) ->
```

- introduire une nouvelle variable de type α
- ajouter $(x : \alpha)$ dans l'environnement
- typer le corps de la fonction `e` (type résultat τ)
- collecter les contraintes sur α
  - Si une solution $\alpha = \tau'$ renvoyer $\tau' \rightarrow \tau$
  - Sinon erreur de typage

```
| App(f,e) -> let tyf = type_expr f
              and tye = type_expr e in
            (* vérifier que tyf de la forme Tfun(tye,tyr),
              renvoyer tyr
             *)
```

Algorithme de typage : Exemples

Appliquer l'algorithme de typage aux expressions:

$$\textbf{fun } x \rightarrow \textbf{fun } y \rightarrow 2 * x < y$$

$$\textbf{fun } f \rightarrow f\ 2 + f\ 3$$

$$\textbf{fun } f \rightarrow f\ 2 + f\ \texttt{true}$$

Objets récursifs

Dans des déclarations locales ou globales, possibilité d'introduire des objets récursifs:

```
let rec f = e
let rec f = e1 in e2
```

Règles de typage:

$$\frac{\rho + (f : \tau) \vdash e_1 : \tau \quad \rho + (f : \tau) \vdash e_2 : \tau_2}{\rho \vdash \texttt{LetRIn}(f, e_1, e_2) : \tau_2}$$

Inférence:

- Introduire pour τ une nouvelle variable de type et résoudre les contraintes.

Fonctions récursives

En pratique dans Ocaml, les définitions récursives sont limitées aux définitions de fonctions `let rec f x = e` qui est la même chose que:
`let rec f = fun x → e`

$$\frac{\rho + (f : \tau \rightarrow \tau')(x : \tau) \vdash e_1 : \tau' \quad \rho + (f : \tau \rightarrow \tau') \vdash e_2 : \tau_2}{\rho \vdash \texttt{LetRIn}(f, \textbf{fun}\ x \ \rightarrow \ e_1, e_2) : \tau_2}$$

On introduit deux variable α et β pour τ et τ'.

Fonctions récursives : exemple

```
let rec fact1 = fun n ->
    if n <= 0 then 1
    else n * fact1 (n-1)
```

Sommaire

Contraintes de type

- Les types ont une structure de termes du premier ordre:

```
type fot = Var of ident | Term of ident * fot list
```

  deux termes sont égaux s'ils sont structurellement égaux
- La résolution des contraintes de type est un problème d'unification du premier ordre:
  - Soient deux termes *t* et *u* avec des variables,
  - on cherche s'il existe une manière de remplacer les variables de *t* et de *u* par d'autres termes qui rend les termes résultant égaux.
- L'association de termes à des variables s'appelle une substitution
- Une substitution peut être appliquée à un terme pour obtenir un nouveau terme

```
let rec subst s = fun
    Var x -> List.assoc x s
  | Term(f,l) -> Term(f, List.map (subst s) l)
```

Exemple

$$
\begin{aligned}
\sigma \quad & = (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta\ \texttt{list} \rightarrow \alpha \\
s \quad & = \{\alpha \mapsto \texttt{bool}, \beta \mapsto \alpha \times \texttt{int}\} \\
s(\sigma) \quad & = (\texttt{bool} \rightarrow \alpha \times \texttt{int} \rightarrow \texttt{bool}) \\
& \rightarrow \texttt{bool} \rightarrow (\alpha \times \texttt{int})\ \texttt{list} \rightarrow \texttt{bool}
\end{aligned}
$$

Ordre sur les termes

- Un terme avec variables représente un ensemble de termes : tous les termes obtenus par substitution
- Les termes peuvent être ordonnés $t_2 \preceq t_1$ s'il existe une substitution s telle que $t_2 = s(t_1)$.
  t_2 est alors moins général que t_1.

Exercice ordonner les types suivants:

- $\alpha \rightarrow \alpha$
- $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$
- $\mathtt{int} \rightarrow \mathtt{int}$
- $\alpha \rightarrow \mathtt{int}$

Unificateur principal

Soient deux termes t et u

- On peut décider s'il existe une substitution s telle que $s(t) = s(u)$
- Si $s(t) = s(u)$ alors pour toute substitution s_0 on a $s_0 \circ s(t) = s_0 \circ s(u)$.
- Si une substitution s telle que $s(t) = s(u)$ existe, il en existe une principale.
  s est principale si pour toute substitution s', si $s'(t) = s'(u)$ alors il existe s_0 tel que $s' = s_0 \circ s$.

Unification-exemples

Unificateur principal

- $\texttt{int}$ et α
  $\{\alpha \mapsto \texttt{int}\}$
- $\texttt{int}$ et $\texttt{bool}$
  Echec
- α et β
  $\{\alpha \mapsto \beta\}$
- α et $\beta\ \texttt{array}$
  $\{\alpha \mapsto \beta\ \texttt{array}\}$
- α et $\alpha\ \texttt{array}$
  Echec (pas de solution finie)

Algorithme d'unification

Algorithme naif : on traite un ensemble d'équations entre termes, on renvoie la substitution résultat

```
let rec unif = fun
   [] -> []
 |(Var x, Var y)::eqs when x = y -> unif eqs
 |(Var x, t)::_ when occur_term x t -> raise UnifError
 |(Var x, t)::eqs -> (x,t)::unif (subst_eqs (x,t) eqs)
 |(Term(f1,lt1),Term(f2,lt2)::_
      when f1<>f2 || List.length lt1 <> List.length lt2
      -> raise UnifError
 |(Term(f1,lt1),Term(f2,lt2)::eqs -> unif (List.combine lt1 lt2@ec
```

- critère de terminaison complexe
- autres algorithmes plus efficaces

Retour sur le typage

- On se donne un terme t à typer
- On résoud les contraintes de type à l'aide d'unificateurs principaux
- On obtient un type qui peut contenir des variables
- C'est le type le plus général (tout autre type valide est une instance)

Exemple 1

Typer **fun** $n \rightarrow$ **fun** $f \rightarrow$ **fun** $x \rightarrow f\ (n\ f\ x)$

1. $n : \alpha, f : \beta, x : \gamma \vdash f : \beta_1 \rightarrow \beta_2$
   Solution $\{\beta = (\beta_1 \rightarrow \beta_2)\}$
2. $n : \alpha, f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash n\ f\ x : \beta_1$
   1. $n : \alpha, f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash n\ f : \alpha_1 \rightarrow \beta_1$
      1. $n : \alpha, f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash n : \alpha_2 \rightarrow (\alpha_1 \rightarrow \beta_1)$
         Solution $\{\alpha = (\alpha_2 \rightarrow (\alpha_1 \rightarrow \beta_1))\}$
      2. $n : \alpha_2 \rightarrow (\alpha_1 \rightarrow \beta_1), f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash f : \alpha_2$
         Solution $\{\alpha_2 = (\beta_1 \rightarrow \beta_2)\}$
   2. $n : (\beta_1 \rightarrow \beta_2) \rightarrow (\alpha_1 \rightarrow \beta_1), f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash x : \alpha_1$
      Solution $\{\gamma = \alpha_1\}$
3. $n : (\beta_1 \rightarrow \beta_2) \rightarrow (\alpha_1 \rightarrow \beta_1), f : \beta_1 \rightarrow \beta_2, x : \alpha_1 \vdash f\ (n\ f\ x) : \beta_2$

Exemple 2

L'expression **fun** $x \rightarrow (x\ x)$ n'est pas typable.
$x : \alpha \vdash x\ x : ?$

1. $x : \alpha \vdash x : \beta_1 \rightarrow \beta_2$
   Solution : $\{\alpha = \beta_1 \rightarrow \beta_2\}$
2. $x : \beta_1 \rightarrow \beta_2 \vdash x : \beta_1$
   Solution ? $\beta_1 \rightarrow \beta_2 \simeq \beta_1$
   Echec : pas de solution finie!

Remarque

- On a introduit des variables dans la structure des types du langage pour pouvoir résoudre les équations.
- Après la résolution, que faire s'il y a encore des variables ?
  - choisir une instance particulière ?
  - échouer ?
- Le polymorphisme est une extension du typage qui permet de profiter de la généricité des opérations.

Sommaire

Polymorphisme

- Certaines constantes sont polymorphes:
  - conditionnelle : `bool` $\rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$
  - accès dans un tableau α `array` $\rightarrow$ `int` $\rightarrow \alpha$
  - listes [] : α `list` et `::` de type $\alpha \rightarrow \alpha$ `list` $\rightarrow \alpha$ `list`
  - ...
- A chaque utilisation d'une de ces constantes, on peut utiliser ce type avec des variables fraiches.

Exemples

```
1::[]
true::[]
1::true::[]
```

Règles de typage polymorphe

- Lors du typage, certaines variables de type restent indéterminées.

```
let singl x = x::[]
```

- On souhaite également les utiliser de manière polymorphe

```
singl 1;;
singl true;;
```

- Intérêt du polymorphisme
  - généralité du code, concision
  - réutilisabilité (bibliothèques)

Généralisation

Les variables de type non déterminées sont généralisées.

$$\rho \vdash \textbf{fun } x \rightarrow x : \forall\alpha.\alpha \rightarrow \alpha$$

Schéma de type: expression de type dont certaines variables sont quantifiées universellement en tête de l'expression: $\forall\alpha.\alpha \rightarrow \alpha$.

Remarques

- α est une variable liée: $\forall\alpha.\alpha \rightarrow \alpha$ représente le même schéma que $\forall\beta.\beta \rightarrow \beta$.
- Une variable qui n'est pas liée est dite libre.

Schémas de type : vocabulaire

Type: schéma de type sans quantificateur.

Type monomorphe: type ne contenant pas de variable de type.

Type polymorphe: schéma de type.

Instancier un schéma de type: remplacer les variables universellement quantifiées par une expression de type.
Un schéma de type représente un ensemble de types (toutes les instanciations possibles).

Généraliser un type: quantifier universellement les variables d'un type pour obtenir un schéma de type.

Représentation possible: un type avec variables + une liste de variables généralisées.

Les règles de typage

- on associe un type (sans quantificateurs) à chaque expression
- Les constantes mais aussi les variables dans l'environnement sont associées à des schéma de types

$\forall \alpha_1 \ldots \alpha_n.typ$
instancier chaque α_i par une *nouvelle* variable de type.
On modifie la règle de typage des constantes/variables.

$$\frac{\texttt{type\_cte}(c) = \forall \alpha_1, ..., \alpha_n.\tau \quad \alpha_i \text{ fraîches}}{\rho \vdash \texttt{Cte}(c) : \tau}$$

$$\frac{(x : \forall \alpha_1, ..., \alpha_n.\tau) \in \rho \quad \alpha_i \text{ fraîches}}{\rho \vdash \texttt{Var}(x) : \tau}$$

Les règles de typage

Exemple

```
let id = fun x -> x
(id true, id 1)
```

$$\dfrac{\dfrac{(\mathtt{id} : \forall\alpha.\alpha \to \alpha) \in \mathit{Env}}{\mathit{Env} \vdash \mathtt{id} : \mathtt{bool} \to \mathtt{bool}} \qquad \mathit{Env} \vdash \mathtt{true} : \mathtt{bool}}{\mathit{Env} \vdash \mathtt{id\ true} : \mathtt{bool}}$$

```
(fun f -> f true, f 1) (fun x -> x)
```

Les règles de typage

Quand et comment généraliser ?

Au moment des déclarations locales et globales

$$\texttt{let}\ \mathit{idf} = \mathit{exp}$$
$$\texttt{let}\ \mathit{idf} = \mathit{exp}1\ \texttt{in}\ \mathit{exp}2$$

On voudrait modifier un peu la règle de typage des déclarations.

$$\frac{\rho \vdash e_1 : \tau_1 \qquad \rho + (x : \forall(\alpha_i)_{i=1..n}.\tau_1) \vdash e_2 : \tau_2}{\rho \vdash \mathtt{Letin}(x, e_1, e_2) : \tau_2}$$

Limites de la généralisation

```
fun x → let f y = [y] in f (f x)
fun x → let f y = [x;y] in f (f x)
```

- Seules les variables non libres dans l'environnement peuvent être généralisées.

Traits impératifs : exemples

```
let counter = ref 0
val counter : int ref = {contents = 0}
let add_counter n =
  counter := !counter + n
val add_counter : int -> unit = <fun>
let afficher mess =
  print_string mess;
  print_newline ()
val afficher : string -> unit = <fun>
exception Division_par_zero
let division x y =
  if y = 0 then raise Division_par_zero
  else x / y}
val division : int -> int -> int = <fun>
```

Traits impératifs : typage

Référence (`ref` *exp*)

$$\frac{Env \vdash exp : \tau}{Env \vdash \texttt{ref}\ exp : \tau\ \texttt{ref}}$$

Affectation (*exp*1 `:=` *exp*2)

$$\frac{Env \vdash exp1 : \tau\ \texttt{ref} \quad Env \vdash \texttt{exp2} : \tau}{Env \vdash exp1 := exp2 : \texttt{unit}}$$

Séquence (*exp*1 `;` *exp*2)

$$\frac{Env \vdash exp1 : \tau_1; \quad Env \vdash exp2 : \tau_2}{Env \vdash exp1; exp2 : \tau_2}$$

Exceptions (`exception` *idf* `of` *typ*)

$$Env \vdash idf : typ \rightarrow \texttt{exn}$$

$$Env \vdash \texttt{raise} : \forall\alpha.\texttt{exn} \rightarrow \alpha$$

Généralisation et effets de bord

Problème dit des références polymorphes
Sans restriction, le programme suivant serait bien typé alors qu'il produit une erreur de type à l'exécution:

```
let lr = ref []
let add x =
  lr := x :: !lr;!lr

add 1; add 2;
add true;;
```

Généralisation et effets de bord

- Une expression est expansive si elle est de la forme $e_1\ e_2$.
- Le type d'une expression expansive n'est pas généralisé.
- La généralisation du type des seules expressions *non expansives* garantit la correction du typage: "tout programme bien typé ne fait pas d'erreur de type à l'exécution".

```
let lr = ref []
val lr : '_a list ref = {contents = []}
let add x =  lr := x :: !lr;!lr
val add : '_a -> '_a list = <fun>
add 1;;
- : int list = [1]
lr;;
- : int list ref = {contents = [1]}
add;;
- : int -> int list = <fun>
add true;;
This expression has type bool but is here used with type int
```

Généralisation et effets de bord

- compromis pour pouvoir définir des fonctions génériques sur les références sans casser le typage

```
let id = fun x -> x
val id : 'a -> 'a = <fun>
let f = id id
val f : '_a -> '_a = <fun>
let g = fun x -> (id id) x
val g : 'a -> 'a = <fun>
```

- d'autres solutions ont été proposées pour résoudre ce problème
- la généralisation des types des expressions non expansives est un critère simple pour l'utilisateur

Extension du polymorphisme

Un schéma de type est un type.
Possibilité de généraliser le type des paramètres des fonctions:

$$(\forall\alpha.\tau) \to \sigma$$

$$\frac{x : \forall\alpha.\sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

modifie profondément le système de type.
Ainsi on a :

$$\frac{x : \forall\alpha.\alpha \to \alpha \vdash x : (\beta \to \beta) \to \beta \to \beta \quad x : \forall\alpha.\alpha \to \alpha \vdash x : \beta \to \beta}{\vdash \textbf{fun}\ x \to (x\ x) : \forall\beta(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta}$$

Système connu sous le nom de lambda-calcul du second ordre ou encore lambda-calcul polymorphe ou encore système F.
Il a été montré que l'inférence de type dans ce système n'était pas décidable.

Polymorphisme de ML

- ML adopte un polymorphisme limité
- On ne généralise que les schémas des variables introduites dans les **let** (et pas les paramètres de fonctions)
- Il s'agit d'un polymorphisme paramétrique: le code est le même quelque soit le type manipulé.

Conclusion

- Déclaration du type des identificateurs/synthèse du type
- Typage fort: une expression a un type préservé par calcul
- Polymorphisme : description unique d'un ensemble de types
- Garanties apportées par le typage fort

Conclusion du cours

Résumé

Ce que l'on a vu dans ce cours

- La compilation est au cœur de la programmation et par conséquent de l'informatique.
- La compilation met en œuvre des notions avancées (théorie, structures de données et algorithmes) :
    - théorie des langages formels: automates finis, automates à pile
    - sémantique des langages de programmation: typage, inférence, évaluation
    - optimisation de code: manipulation de graphes, architecture des ordinateurs

Ce qu'il faut retenir

- La distinction entre ce qui peut être traité par l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.
- La correspondance entre les langages reconnus et les expressions régulières et les grammaires.
- Le fonctionnement de l'analyse descendante et de l'analyse ascendante (en particulier la notion de lecture, réduction et les conflits qui en découlent).
- Le rôle des règles de précédence dans la résolution des conflits.
- La mise en œuvre d'un analyseur lexical pour les transformations simples de texte.

Ce qu'il faut retenir

- La description d'un arbre de syntaxe abstraite.
- L'analyse de portée (la construction et la représentation de la table des symboles).
- Le rôle du typage, les règles de typage simple (les comprendre, les mettre en œuvre).
- L'analyse de flots de données.

Ce qu'il faut retenir

- Les constructions de base de l'assembleur (opérations arithmétiques, branchements, accès mémoire)
- La compilation des procédures à l'aide de tableaux d'activation.
- La représentation en machine de données complexes (structures, objets, fonctions).
- Les langages intermédiaires de type RTL.
- La problématique de l'allocation de registres.