Typage et Analyse Statique

Cours 2

Emmanuel Chailloux

Spécialité Science et Technologie du Logiciel
Master mention Informatique
Université Pierre et Marie Curie

année 2016-2017

Plan du cours 2

Polymorphisme paramétrique:

- λ-calcul typé
  - types simples
    - vérification d'un type
    - unification
    - synthèse de types
  - schémas de types et polymorphisme
- typage d'un mini-ML
  - noyau fonctionnel
  - extension impérative

Préambule

On a vu que le λ-calcul est la pure théorie de la fonctionalité. Mais on peut vouloir associer à chaque variable d'un terme un *type* qui indique sa nature fonctionnelle, c'est-à-dire ses domaine et codomaine. C'est naturel quand on pense aux mathématiques où on s'intéresse rarement à une fonction indépendamment de son ensemble de départ et de son ensemble d'arrivée. Les types permettent de retrouver cette idée dans le λ-calcul.

De plus, ils ont un intérêt calculatoire, car l'intérêt d'un système de types pour le λ-calcul (d'un point de vue théorique) est d'assurer la forte normalisation des termes typables. Ce sera le cas du système que nous allons voir, qui est le plus simple de tous. C'est pourquoi on l'appelle le système des types simples. Dans cette optique, le λ-calcul est vu comme un langage de programmation, et les types assurent la terminaison des programmes typables (indépendamment de la stratégie de réduction).

Types simples

La syntaxe du λ-calcul est identique à celle qui a été introduite précédemment.

Soit $\mathcal{P}$ un ensemble de variables de types. On définit l'ensemble des types simples bien formés $\mathcal{T}$ (construit à partir de $\mathcal{T}$ et du constructeur $\rightarrow$) par :

- si $\alpha \in \mathcal{P}$ alors $\alpha \in \mathcal{T}$
- si $\sigma, \tau \in \mathcal{T}$ alors $\sigma \rightarrow \tau \in \mathcal{T}$.

exemples:
$(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$
$(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$

Contexte de typage

Typer un terme (clos), c'est lui associer un type. Mais on ne peut pas se contenter de ne typer que les termes clos. Il faut donc se référer à un *contexte de types* (qui associe à chaque variable libre un type).

Donc typer un terme (non nécessairement clos) consistera à lui associer un type dans un certain contexte où se trouvent toutes les variables libres du terme. Donnons une définition plus formelle.

Les contextes sont des listes de couples $C = (x_1 : \sigma_1), \ldots, (x_n : \sigma_n)$.

typer un terme

On définit inductivement une relation $C \vdash M : \tau$ qui signifie "M est de type τ dans le contexte C" de la façon suivante :

- Si $C = (x_1 : \sigma_1), \ldots, (x_n : \sigma_n)$, alors $C \vdash x_i : \sigma_i$ pour tout $i = 1, \ldots, n$ (Var).
- Si $C \vdash M : \sigma \to \tau$ et $C \vdash N : \sigma$ alors $C \vdash MN : \tau$ (App).
- Et si $(x : \sigma)C \vdash M : \tau$ alors $C \vdash \lambda x.M : \sigma \to \tau$ (Abs).

Ces règles s'écrivent habituellement :

(Var)
$$(x_1 : \sigma_1), \ldots, (x_n : \sigma_n) \vdash x_i : \sigma_i$$

(App)
$$\frac{C \vdash M : \sigma \to \tau \quad C \vdash N : \sigma}{C \vdash MN : \tau}$$

(Abs)
$$\frac{(x : \sigma), C \vdash M : \tau}{C \vdash \lambda x.M : \sigma \to \tau}$$

Exemple de typage (1)

on se propose de typer l'entier de Church "deux"($\lambda f \lambda x.f(fx)$). On prouve qu'il admet le type $(\alpha \to \alpha) \to \alpha \to \alpha$ au moyens des règles ci-dessus.

1) on a $(x : \alpha), (f : \alpha \to \alpha) \vdash x : \alpha$
2) et $(x : \alpha), (f : \alpha \to \alpha) \vdash f : \alpha \to \alpha$
3) donc par 1) et 2) $(x : \alpha), (f : \alpha \to \alpha) \vdash fx : \alpha$
4) par 2) et 3) on a $(x : \alpha), (f : \alpha \to \alpha) \vdash f(fx) : \alpha$
5) puis par la règle d'abstraction, de 4) on déduit $(f : \alpha \to \alpha) \vdash \lambda x.f(fx) : \alpha \to \alpha$
6) et de même $(f : \alpha \to \alpha) \vdash \lambda f.\lambda x.f(fx) : (\alpha \to \alpha) \to \alpha \to \alpha$

Exemple de typage (2)

Ce qui s'écrit plus facilement en utilisant la deuxième forme des règles :

$$\dfrac{\dfrac{x : \alpha, f : \alpha \to \alpha \vdash f : \alpha \to \alpha \quad \dfrac{x : \alpha, f : \alpha \vdash f : \alpha \to \alpha \quad x : \alpha, f : \alpha \to \alpha \vdash x : \alpha}{x : \alpha, f : \alpha \to \alpha \vdash fx : \alpha}}{x : \alpha, f : \alpha \to \alpha \vdash f(fx) : \alpha}}{f : \alpha \to \alpha \vdash \lambda x.f(fx) : \alpha \to \alpha}}{\vdash \lambda f.\lambda x.f(fx) : (\alpha \to \alpha) \to \alpha \to \alpha}$$

On a donné une méthode de *vérification* de typage, mais pas de *synthèse* d'un type.

Synthèse

On a donné une méthode de *vérification* de typage, mais pas de *synthèse* d'un type.

Pour ce système de types, il existe une méthode de synthèse utilisant l'unification, que nous allons donner.

En voici un avant-goût par le raisonnement intuitif suivant : On suppose que dans ce terme, f a le type σ et x a le type τ.

Comme on a pu former fx il faut que σ soit de la forme $\tau \to \sigma_1$ et alors fx est du type σ_1. Comme on a pu former $f(fx)$, il faut que $\tau = \sigma_1$. En conclusion, $f : \tau \to \tau$ et $x : \tau$, et le terme a le type $(\tau \to \tau) \to \tau \to \tau$ pour n'importe quel type τ (en particulier, on peut prendre une variable α).

Synthèse par unification

On va rappeler l'algorithme d'unification, dans le cas simple qui est le nôtre. On considère une signature ne comportant qu'un seul symbole, une fonction a d'arité 2 (c'est la flèche $\rightarrow$ des types). Les variables que nous considérerons sont celles de l'ensemble $\mathcal{P}$.
Rappelons qu'une substitution est une application $\mathcal{P} \rightarrow \mathcal{T}$ ($\mathcal{T}$ est l'ensemble des termes bâtis sur cette signature), et qu'on peut appliquer une substitution s à un terme $\sigma \in \mathcal{T}$, ce qui donne un terme σs :

- $xs = s(x)$
- $(a(\sigma, \tau))s = a(\sigma s, \tau s)$

Cela permet de définir la composée $s \circ t$ de deux substitutions comme une substitution.
Le problème consiste à résoudre des équations dans cette signature, et de trouver la solution la plus générale possible. Pour donner un sens à cela, il faut mettre un ordre sur les substitutions.

Unification (1)

Définition

On dit que s est plus général que t s'il existe une substitution u telle que $u \circ s = t$.

Si $\sigma, \tau \in \mathcal{T}$, unifier σ et τ, c'est trouver une substitution s telle que $\sigma s = \tau s$. Alors le résultat est le suivant :

Théorème

Pour tous $\sigma, \tau \in \mathcal{T}$ unifiable, il existe un unificateur qui est plus général que tous les autres. On l'appelle unificateur principal, et on le note $UP(\sigma, \tau)$.

La preuve de ce théorème ne présente d'intérêt que par l'algorithme d'unification qu'elle propose, qui calcule cet unificateur principal ou échoue. Nous rappelons à présent cet algorithme. On remarquera qu'ici la seule cause d'échec possible est celle qui résulte du test d'occurrence d'une variable.

Unification (2)

On notera $[\sigma/\alpha]$ la substitution qui envoie α sur σ et toutes les autres variables sur elles-mêmes. Alors, on définit la fonction UP qui prend deux termes et rend une substitution inductivement par :

- $UP(\alpha, \sigma) = [\sigma/\alpha]$ si α ne figure pas dans σ, et échec si non.
- $UP(\sigma, \alpha) = [\sigma/\alpha]$ si α ne figure pas dans σ, et échec si non.
- si $s = UP(\sigma, \sigma')$ et $t = UP(\tau s, \tau' s)$, alors $UP(a(\sigma, \tau), a(\sigma', \tau') = t \circ s)$.

Justification de l'algorithme (ce n'est pas une preuve rigoureuse). Le seul cas non trivial est celui où l'on a à unifier $a(\sigma_1, \tau_1)$ avec $a(\sigma_2, \tau_2)$. Alors si $s = UP(\sigma_1, \sigma_2)$, alors $\sigma_1 s = \sigma_2 s$ et si $t = UP(\tau_1 s, \tau_2 s)$, alors $(\tau_1 s)t = (\tau_2 s)t$, c'est-à-dire $\tau_1 u = \tau_2 u$ où $u = t \circ s$. On voit facilement que u est un unificateur de nos deux termes de départ. Il est principal car sinon il existerait une substitution, non contenue dans $(t; s)$, qui composée à lui atteindrait l'UP du terme, ce qui est contradictoire avec sa construction.

Implantation

Voici l'algorithme d'unification dans le cas des types simples. La syntaxe abstraite des types :

```
1  type oType =
2    Tvar of string
3  | Arrow of oType * oType;;
```

On représente les substitutions par des listes de couples chaîne de caractère, terme, Voici l'application d'une substitution à un type :

```
1  (*  val substitute : (string * oType) list -> oType -> oType = <fun> *)
2
3  let substitute s t =
4    let rec  sub_rec t = match t with
5       (Tvar a) as alpha -> (try List.assoc a s with Not_found -> alpha)
6     | Arrow(oT1,oT2)    -> Arrow(sub_rec oT1, sub_rec oT2)
7    in sub_rec t
8  ;;
```

et voici la composition de deux substitutions :

Algorithme de typage

On définit une fonction T qui prend un contexte et un terme M et rend un couple constitué d'une substitution et du type cherché pour M dans ce contexte (on produit le type le plus général possible).

- $T(x, C) = (C(x), id)$
- Pour calculer $T(\lambda x.M, C)$, on introduit une nouvelle variable de type α, et on essaie de typer M dans le nouveau contexte $(x : \alpha)C$; soit $(\sigma, s) = T(M, (x : \alpha)C)$. Alors le type cherché est $s(\alpha) \to \sigma$, et on rend la substitution s.
- Pour calculer $T(MN, C)$, on calcule $(\sigma, s) = T(M, C)$, et $(\tau, t) = T(N, Cs)$, car il faut tenir compte des contraintes induites par le typage de M. On veut pouvoir appliquer M à N, et pour cela il faut que M ait un type fonctionnel dans le contexte courant, c'est-à-dire que σt soit de la forme $\tau \to \phi$ où ϕ est un type à déterminer. Pour cela, on calcule l'unificateur principal u de σt et $\tau \to \alpha$ où α est une nouvelle variable de type. On rend le type $u(\alpha)$, et la substitution $u \circ t \circ s$.

Implantation (1)

Voici le programme de typage en Caml :
Le type `term` pour la représentation des λ-termes et la fonction `gensym` pour la création des variables de type sont définis ainsi :

```
type term= Var of string
     | Abs of string * term
     | App of term * term;;
```

```
# let c = ref 0;;
val c : int ref = {contents = 0}
# let gensym s =  (c:=!c+1 ;s^(string_of_int !c));;
val gensym : string -> string = <fun>
# let reset () = c := 0;;
val reset : unit -> unit = <fun>
```

Implantation (2)

La fonction suivante TYPE suit très exactement l'algorithme de typage indiqué ci-dessus.

```
1  (*  val oTYPE : (string * oType) list -> term -> oType * (string * oType) ↩
       list =   <fun> *)
2
3  let rec oTYPE oC t = match t with
4       Var x    -> List.assoc x oC , []
5    |  Abs(x,oM) ->
6         let alpha=gensym "a" in
7           let (sigma,s) = oTYPE ((x,Tvar alpha)::oC) oM in
8             Arrow( (try List.assoc alpha s with _ ->  Tvar alpha),
9                           sigma), s
10   |  App(oM,oN) ->
11        let (sigma,s)=oTYPE oC oM in
12          let (tau,t) = oTYPE (List.map (function (x,phi) ->
13                                  (x,substitute s phi)) oC) oN   in
14            let alpha=gensym "a" in
15              let u= unify(substitute t sigma, Arrow(tau,Tvar alpha))  in
16                (try List.assoc alpha u with _ ->  Tvar alpha),
17                           compsubst u (compsubst t s)
18 ;;
```

Implantation (3)

La fonction `print_type` donne un affichage lisible des types.

```
(*   val print_type : oType -> unit = <fun> *)
let print_type t =
  let rec ptype t = match t with
    Tvar x -> print_string x
  | Arrow (x,y) -> print_string "("; ptype x;print_string " -> ";
                   ptype y;print_string ")"
  in
    ptype t;print_newline() ;;
```

Les exemples suivants reprennent des termes déjà définis.

1. A :

```
# let oA = Abs ( "x" , Abs ("y" , App (Var "x", Var "y") ));;
val oA : term = Abs ("x", Abs ("y", App (Var "x", Var "y")))
# oTYPE [] oA;;
- : oType * (string * oType) list =
(Arrow (Arrow (Tvar "a2", Tvar "a3"), Arrow (Tvar "a2", Tvar "a3")),
 [("a1", Arrow (Tvar "a2", Tvar "a3"))])
# print_type (fst (oTYPE [] oA));;
((a5 -> a6) -> (a5 -> a6))
- : unit = ()
```

Implantation (4)

1. S :

```
# let oS = Abs(  "x" ,
            Abs ( "y" ,
                Abs ( "z", App ( App (Var "x" , Var "z" ) ,
                                    App (Var "y",Var "z") ) )));;
val oS : term =
  Abs ("x",
   Abs ("y", Abs ("z", App (App (Var "x", Var "z"), App (Var "y", Var "z")))))
# print_type (fst (oTYPE [] oS));;
((a9 -> (a11 -> a12)) -> ((a9 -> a11) -> (a9 -> a12)))
- : unit = ()
```

2. Δ :

```
# let delta = Abs ( "x" , App (Var "x" , Var "x" ));;a
val delta : term = Abs ("x", App (Var "x", Var "x"))
#  print_type (fst (oTYPE [] oS));;
((a17 -> (a19 -> a20)) -> ((a17 -> a19) -> (a17 -> a20)))
- : unit = ()
# print_type (fst (oTYPE [] delta));;
Exception: Failure "unify".
```

Propriété

On dit qu'un terme M est typable (avec les types simples) s'il existe un contexte C et un type σ tels que $C \vdash M : \sigma$. Voici le théorème :

Théorème

Tout terme typable (avec les types simples) est fortement normalisable.

On n'en donnera pas la preuve.

Les termes comme Ω ou Y (point fixe) ne sont pas typables dans le système de types simples.

Polymorphisme (1)

Le système des types simples ne donne pas le vrai polymorphisme malgré son ensemble de variables de type. En effet le terme $(\lambda f.ff)(\lambda x.x)$ n'est pas typable car la variable f prend deux valeurs : $\alpha \to \alpha$ et $(\alpha \to \alpha) \to \alpha \to \alpha$. Ce qui est impossible car une variable ne possède qu'un seul type.
Pour cela on étend le système de type. Soit $\mathcal{P}$ un ensemble de variables de types, $\mathcal{T}$ l'ensemble des types simples construit à partir de $\mathcal{T}$ et $\mathcal{S}$ l'ensemble des schémas de types construit à partir de $\mathcal{P}$ et $\mathcal{T}$.

- types simples :
  - si $\alpha \in \mathcal{P}$ alors $\alpha \in \mathcal{T}$
  - si $\sigma, \tau \in \mathcal{T}$ alors $\sigma \to \tau \in \mathcal{T}$.
- schémas de type :
  - si $\tau \in \mathcal{T}$ alors $\tau \in \mathcal{S}$
  - si $\sigma \in \mathcal{S}, \alpha \in \mathcal{P}$ alors $\forall \alpha.\sigma \in \mathcal{S}$

Polymorphisme (2)

On obtient ainsi de nouvelles règles de typage :

(Var)

$$(x_1 : \sigma_1), \ldots, (x_n : \sigma_n) \vdash x_i : \tau[\tau_i/\alpha_i] \quad \sigma_i = \forall \alpha_1, \ldots, \alpha_n.\tau$$

(App)

$$\frac{C \vdash M : \tau \to \tau\prime \quad C \vdash N : \tau}{C \vdash MN : \tau\prime}$$

(Abs)

$$\frac{(x : \tau), C \vdash M : \tau\prime}{C \vdash \lambda x.M : \tau \to \tau\prime}$$

(Let)

$$\frac{C \vdash N : \tau \quad \alpha_1, \ldots, \alpha_n = V(\tau) - V(C) \quad (x : \forall \alpha_1, \ldots, \alpha_n.\tau), C \vdash M : \tau\prime}{C \vdash \mathit{let}\ x\ =\ N\ \mathit{in}\ M : \tau\prime}$$

Ce qui nous permet de typer `let f = λx.x in ff` :

$$\frac{\dfrac{\dfrac{}{f : \forall\beta.\beta \to \beta \vdash f : (\alpha \to \alpha) \to \alpha \to \alpha}\,VAR \quad \dfrac{}{f : \forall\beta.\beta \to \beta \vdash f : \alpha \to \alpha}\,VAR}{\vdash \lambda x.x : \beta \to \beta \quad \beta = V(\beta \to \beta) - V(\emptyset) \quad f : \forall\beta.\beta \to \beta \vdash ff : \alpha \to \alpha}}{\vdash \mathit{let}\ f\ =\ \lambda x.x\ \mathit{in}\ ff : \alpha \to \alpha}$$

du λ-calcul aux langages de programmation (1)

Langage d'expressions : λ-calcul auquel on ajoute :

- des constantes : entiers, booléens
- des opérateurs arithmétiques et logiques
- une stratégie d'évaluation : immédiate (ou retardée)
- une forme spéciale pour la conditionnelle (ITE)
- nommage d'une expression (LET IN) avec portée statique

du λ-calcul aux langages de programmation (2)

- déclaration récursive (LETREC IN)
- structures de données :
  - couples : constructeur et accesseurs (fst, snd)
  - listes : constructeurs (`[]`, `::`), et accesseurs (hd, tl)

On peut y ajouter les déclarations globales, et bien sûr un système de types.

mini-ML

Un compilateur ML se décompose de manière classique en :

- une analyse lexicale puis syntaxique du texte source en entrée qui construit un arbre de syntaxe ;
- un vérificateur de types qui infère les types des expressions et détecte donc les erreurs de typage ;
- un producteur de code assembleur, souvent d'une machine abstraite ;
- et d'une phase d'assemblage pour exécuter ce code sur un processeur existant.

On se propose dans cette section de définir la syntaxe abstraite d'un langage appelé mini-ML,

mini-ML

Mini-ML est uniquement un langage d'expressions. Elles correspondent à une extension du λ-calcul par les constantes numériques, de chaînes de caractères et des booléens, des couples et des listes, de la structure de contrôle conditionnelle, de la déclaration locale `let` qui introduit le polymorphisme et de la récursion.

```
1  type ml_expr = Const of ml_const
2    | Var of string
3    | Pair of ml_expr * ml_expr
4    | Cons of ml_expr * ml_expr
5    | Cond of ml_expr * ml_expr * ml_expr
6    | App of ml_expr * ml_expr
7    | Abs of string * ml_expr
8    | Letin of string * ml_expr * ml_expr
9    | Letrecin of string * ml_expr * ml_expr
10 and ml_const = Int of int
11   | Float of float
12   | Bool of bool
13   | String of string
14   | Emptylist
15   | Unit;;
```

Syntaxe des types (1)

L'ensemble des types ($\mathcal{T}$) de mini-ML est construit à partir d'un ensemble de constantes de types ($\mathcal{C}$), un ensemble de variables de types ($\mathcal{P}$) et de 3 constructeurs de types ($\rightarrow$, $*$ et *list*).

- constante de type :
  `Int_type`, `Float_type`, `String_type`, `Bool_type` et `Unit_type` $\in \mathcal{C}$: si $\alpha \in \mathcal{C}$ alors $\alpha \in \mathcal{T}$;
- variable de type $\in \mathcal{P}$:si $\alpha \in \mathcal{P}$ alors $\alpha \in \mathcal{T}$;.
- type produit : si $\sigma, \tau \in \mathcal{T}$ alors $\sigma \ * \ \tau \in \mathcal{T}$;
- type liste : si $\sigma \in \mathcal{T}$ alors σ *list* $\in \mathcal{T}$;
- type fonctionnel : si $\sigma, \tau \in \mathcal{T}$ alors $\sigma \ \rightarrow \ \tau \in \mathcal{T}$;

Syntaxe des types (2)

La syntaxe abstraite des types de mini-ML est la suivante :

```
type vartype =
  | Unknown of int
  | Instanciated of ml_type
and consttype =
  | Int_type
  | Float_type
  | String_type
  | Bool_type
  | Unit_type
and ml_type =
  | Var_type of vartype ref
  | Const_type of consttype
  | Pair_type of ml_type * ml_type
  | List_type of ml_type
  | Fun_type of ml_type * ml_type;;
```

Où une variable de type est une référence sur une inconnue ou un type. Cela permettra d'effectuer une modification de types lors de l'application d'une substitution sur une variable de type.

...

La fonction de création de variables de type est la suivante :

```
let new_unknown,reset_unknowns =
  let c = ref 1 in
    ( (function () -> c:= !c+1; Var_type( ref(Unknown !c))),
      (function () -> c:=1)) ;;
```

L'ensemble des types qui vient d'être décrit correspondant aux types simples étendus par les constantes de type et les constructeurs pour les paires et les listes. Il est encore nécessaire de définir les schémas de type (type `quantified_type`).

```
type quantified_type = Forall of (int list) * ml_type;;
```

Environnement de typage

L'environnement de typage, nécessaire lors du typage pour les variables libres (comme les primitives ou les paramètres formels) d'expressions est représenté par une liste d'association `(string * quantified_type) list`.

Il est nécessaire de faire une distinction entre les variables d'un type si elles sont quantifiées (variables liées) ou si elles ne le sont pas (variables libres).

```
1  let rec vars_of_type t =
2    let rec vars vl = function
3      Const_type _ -> vl
4    | Var_type vt ->
5        ( match !vt with
6            Unknown n -> if List.mem n vl then vl else n::vl
7          | Instanciated t -> vars vl t
8        )
9    | Pair_type (t1,t2) -> vars (vars vl t1) t2
10   | List_type t -> vars vl t
11   | Fun_type (t1,t2) -> vars (vars vl t1) t2
12   in
13     vars [] t ;;
```

Variables libres et liées

Pour calculer les variables libres et liées d'un schéma de type,

```
let subtract l1 l2 =
  List.flatten (List.map (function id ->
                            if (List.mem id l2) then [] else [id])
                 l1);;
let free_vars_of_type (bv,t) =
  subtract (vars_of_type t) bv
and bound_vars_of_type (fv,t) =
  subtract (vars_of_type t) fv ;;
let append l1 l2 = l1@l2;;
let flat ll = List.fold_right append ll [];;
let free_vars_of_type_env l =
    flat ( List.map (function (id,Forall (v,t))
                        -> free_vars_of_type (v,t)) l) ;;
```

Instance d'un schéma de type

Lors du typage d'un identificateur dans un environnement `C`, le vérificateur cherche dans l'environnement le type quantifié associé à cet identificateur et en retourne une instance, où les variables quantifiées valent de nouvelles variables (ou inconnues).

```
1  let type_instance st =
2    match st with Forall(gv,t) ->
3      let unknowns = List.map (function n -> n,new_unknown()) gv
4      in
5        let rec instance = function
6          Var_type {contents=(Unknown n)} as t ->
7             (try List.assoc n unknowns with Not_found -> t)
8        | Var_type {contents=(Instanciated t)} -> instance t
9        | Const_type ct as t -> t
10       | Pair_type (t1,t2) -> Pair_type (instance t1, instance t2)
11       | List_type t -> List_type (instance t)
12       | Fun_type (t1,t2) -> Fun_type (instance t1, instance t2)
13       in
14         instance t ;;
```

Erreurs de typage

Les exceptions suivantes sont levées quand une erreur de type est détectée :

```
1  type typing_error =
2    Unbound_var of string
3  | Clash of ml_type * ml_type ;;
4  exception Type_error of typing_error;;
```

Elles correspondent soit à une variable d'une expression qui n'a pas été déclarée, soit à une erreur proprement dite de typage.

Unification

La grande différence dans l'algorithme de typage du λ-calcul et celui présenté ici provient de l'algorithme d'unification. dans le premier cas, la fonction TYPE retournait la liste des substitutions (l'unificateur principal UP) à appliquer pour que deux termes $t1$ et $t2$ vérifient $UP(t1) = UP(t2)$. Cet algorithme bien que juste n'est pas forcément très efficace. En mini-ML, Les substitutions trouvées sont immédiatement appliquées aux types en effectuant une modification physique de ces types.
Deux problèmes se posent :

- la vérification d'occurrences, quand une variable de type est remplacée par un autre type, ce second type ne doit pas contenir d'occurrence de cette première variable. La fonction `occurs` réalise ce test.
- Lorsqu'une inconnue se voit attribuer comme valeur une autre inconnue, on crée une chaîne d'indirection qu'il faut alors simplifier : `Var_type (ref (Var_type ( ref (Instanciated t))))` deviendra alors seulement `t`.

Fonctions auxiliaires (1)

La fonction shorten réalise ce travail.

```
let occurs n t = List.mem n (vars_of_type t);;
let rec shorten = function
    Var_type (vt) as tt ->
      (match !vt with
           Unknown _  -> tt
         | Instanciated ((Var_type _) as t) ->
             let t2 = shorten t in
               vt := Instanciated t;
               t2
         | Instanciated t -> t
      )
  | t -> t;;
```

Fonctions auxiliaires (2)

La fonction `unify_types` prend deux types, modifie leur taille (par la fonction `shorten`) et les rend égaux ou échoue.

```
let rec unify_types (t1,t2) =
   let lt1 = shorten t1 and lt2 = shorten t2
   in
     match (lt1,lt2) with
     | Var_type ( {contents=Unknown n} as occn ),
       Var_type {contents=Unknown m} ->
         if n=m then () else occn:= Instanciated lt2
     | Var_type ({contents=(Unknown n)} as occn), _ ->
         if occurs n lt2
         then raise (Type_error(Clash(lt1,lt2)))
         else occn:=Instanciated lt2
     | _ , Var_type ({contents=(Unknown n)}) -> unify_types (lt2,lt1)
     | Const_type ct1, Const_type ct2 ->
         if ct1=ct2 then () else raise (Type_error(Clash(lt1,lt2)))
     | Pair_type (t1,t2), Pair_type (t3,t4) ->
         unify_types (t1,t3); unify_types(t2,t4)
     | List_type t1, List_type t2 ->  unify_types (t1,t2)
     | Fun_type (t1,t2), Fun_type (t3,t4) ->
        unify_types (t1,t3); unify_types(t2,t4)
     | _ ->  raise(Type_error(Clash(lt1,lt2)));;
```

Typage des expressions (1)

Le typage des constantes est très simple à l'exception de la liste vide. Ces règles de typage sont toutes considérées comme des axiomes.

(ConstInt)
$$C \vdash \textit{nombre entier} : \textit{Int}$$

(ConstFloat)
$$C \vdash \textit{nombre flottant} : \textit{Float}$$

(ConstString)
$$C \vdash \textit{chaine} : \textit{String}$$

(ConstBool)
$$C \vdash \textit{booleen} : \textit{Bool}$$

(ConstUnit)
$$C \vdash () : \textit{Unit}$$

Typage des expressions (2)

Ces règles se traduisent très simplement en OCaml :

```
1  let type_const = function
2    Int _ -> Const_type Int_type
3  | Float _ -> Const_type Float_type
4  | String _ -> Const_type String_type
5  | Bool _ ->  Const_type Bool_type
6  | Unit ->  Const_type Unit_type
7  | Emptylist -> List_type (new_unknown()) ;;
```

On introduit deux fonctions *instance* qui retourne une instance de type à partir d'un schéma de types et *generalize* qui crée un schéma de types à partir d'un type et d'un environnement.

Instance et généralisation (1)

La fonction *instance* correspond à la fonction OCaml `type_instance`. Elle remplace les variables de types quantifiées d'un schéma de type par de nouvelles variables de types. La fonction *generalize* correspond à la fonction OCaml suivante :

```
let type_instance st =
  match st with Forall(gv,t)  ->
  let unknowns = List.map (function n -> n,new_unknown()) gv
  in
    let rec instance t  = match t with
        Var_type (vt) as tt ->
          ( match !vt with
              Unknown n ->(try List.assoc n unknowns with Not_found -> tt)
            | Instanciated ti -> instance ti
          )
      | Const_type tc  -> t
      | List_type t1 -> List_type (instance t1)
      | Fun_type (t1,t2) -> Fun_type (instance t1, instance t2)
      | Pair_type (t1,t2) -> Pair_type (instance t1, instance t2)
    in
      instance t
;;
```

Instance et généralisation (2)

```
1  let generalize_types gamma l =
2    let fvg = free_vars_of_type_env gamma
3    in
4      List.map (function (s,t) ->
5                  (s, Forall(free_vars_of_type (fvg,t),t))) l
6  ;;
```

Elle crée un schéma de type en quantifiant les variables libres d'un type qui n'appartiennent pas aux variables libres de l'environnement.

Règles de typage (1)

Les règles de typages des expressions sont les règles de typage des constantes et les règles suivantes :

(Var)

$$x : \sigma, C \vdash x : \tau \quad \tau = instance(\sigma)$$

(Pair)

$$\frac{C \vdash e_1 : \tau_1 \quad C \vdash e_2 : \tau_2}{C \vdash (e_1, e_2) : \tau_1 \ * \ \tau_2}$$

(List)

$$\frac{C \vdash e_1 : \tau \quad C \vdash e_2 : \tau \ list}{C \vdash (e_1 \ :: \ e_2) : \tau \ list}$$

(If)

$$\frac{C \vdash e_1 : Bool \quad C \vdash e_2 : \tau \quad C \vdash e_3 : \tau}{C \vdash (if \ e_1 \ then \ e_2 \ else \ e_3) : \tau}$$

(App)

$$\frac{C \vdash e_1 : \tau' \to \tau \quad C \vdash e_2 : \tau'}{C \vdash (e_1 \ e_2) : \tau}$$

(Abs)

$$\frac{x : \tau_1, C \vdash e : \tau_2}{C \vdash (function \ x \ -> \ e) : \tau_1 \to \tau_2}$$

Règles de typage (2)

(Let)

$$\frac{C \vdash e_1 = \tau_1 \quad \sigma = \mathit{generalize}(\tau_1, C) \quad (x : \sigma), C \vdash e_2 : \tau_2}{C \vdash \mathit{let}\ x\ =\ e_1\ \mathit{in}\ e_2 : \tau_2}$$

(LetRec)

$$\frac{(x : \alpha), C \vdash e_1 = \tau_1 \quad \alpha \notin V(C) \quad \sigma = \mathit{generalize}(\tau_1, C) \quad (x : \sigma), C \vdash e_2 : \tau_2}{C \vdash \mathit{letrec}\ x\ =\ e_1\ \mathit{in}\ e_2 : \tau_2}$$

Le polymorphisme est introduit dans les règles *Let* et *LetRec*. La règle *LetRec* suppose que la variable définie récursivement (x) est du type α, nouvelle variable de type qui n'apparaît pas libre dans l'environnement. Ce qui ne fait aucune supposition sur le type de x, mais dans la mesure où le type de x n'est pas un schéma de type, les contraintes de type sur x dans e_1 seront bien répercutées sur le type final de e_1.

Fonction type_expr (1)

La fonction type_expr suivante suit les règles de typage définies ci-dessus.

```
1  let rec type_expr gamma =
2    let rec type_rec expri =
3      match expri with
4          Const c -> type_const c
5        | Var s -> let t = try List.assoc s gamma
6           with Not_found -> raise (Type_error(Unbound_var s))
7           in  type_instance t
8        | Pair (e1,e2) -> Pair_type (type_rec e1, type_rec e2)
9        | Cons (e1,e2) ->
10          let t1 = type_rec e1
11          and t2 = type_rec e2 in
12            unify_types (List_type t1, t2); t2
13       | Cond (e1,e2,e3) ->
14          let t1 = unify_types (Const_type Bool_type, type_rec e1)
15          and t2 = type_rec e2
16          and t3 = type_rec e3 in
17            unify_types (t2,t3); t2
```

Fonction type_expr (2)

```
1       | App (e1,e2) ->
2           let t1 = type_rec e1
3           and t2 = type_rec e2 in
4             let u = new_unknown() in
5               unify_types (t1, Fun_type (t2,u)); u
6       | Abs(s,e) ->
7           let t = new_unknown() in
8             let new_env = (s,Forall ([],t))::gamma in
9               Fun_type (t, type_expr new_env e)
10      | Letin (s,e1,e2) ->
11          let t1 = type_rec e1 in
12            let new_env = generalize_types gamma [ (s,t1) ] in
13              type_expr (new_env@gamma) e2
14      | Letrecin (s,e1,e2) ->
15          let u = new_unknown () in
16            let new_env = (s,Forall([  ],u))::gamma in
17              let t1 = type_expr (new_env@gamma) e1 in
18                let final_env = generalize_types gamma [ (s,t1) ] in
19                  type_expr (final_env@gamma) e2
20  in
21    type_rec;;
```

Environnement initial

```
1   let mk_type (ct1,ct2,ct3) =
2     Forall([],
3      Fun_type (Pair_type(Const_type ct1, Const_type ct2),Const_type ct3)) in
4     let int_ftype = mk_type(Int_type,Int_type,Int_type)
5     and float_ftype = mk_type(Float_type,Float_type,Float_type)
6     and int_predtype = mk_type(Int_type,Int_type,Bool_type)
7     and float_predtype = mk_type(Float_type,Float_type,Bool_type)
8     and alpha = Var_type(ref(Unknown 1))
9     and beta = Var_type(ref(Unknown 2))  in
10    ("=",Forall([ 1 ],
11       Fun_type (Pair_type (alpha,alpha), Const_type Bool_type))::
12    (List.map (function s -> (s,int_ftype)) [ "*"; "+"; "-"; "/" ]) @
13    (List.map (function s -> (s,float_ftype)) [ "*."; "+."; "-."; "/." ]) @
14    (List.map (function s -> (s,int_predtype)) [ "<"; ">"; "<="; ">=" ]) @
15    (List.map (function s -> (s,float_predtype)) [ "<.";">.";"<=.";">=."]) @
16    [ "^", mk_type (String_type, String_type, String_type) ] @
17    [ ("hd",Forall([ 1 ], Fun_type (List_type alpha, alpha)));
18    ("tl",Forall([ 1 ], Fun_type (List_type alpha, List_type alpha)));
19    ("null",Forall([ 1 ], Fun_type (alpha,
20    Fun_type (List_type alpha, Const_type Bool_type))));
21    ("fst",Forall([ 1; 2 ], Fun_type (Pair_type (alpha,beta),alpha)));
22    ("snd",Forall([ 1; 2 ], Fun_type (Pair_type (alpha,beta),beta))) ]  ;;
```

Impression des types et des erreurs (1)

Comme les variables de types sont codées à l'aide d'entiers, il serait particulièrement désagréable d'avoir un affichage de types dépendant de cette numérotation. Pour cela, pour chaque expression globale est crée une liste d'association des numéros de variables de types et des symboles pour les types polymorphes.

Les constantes de types sont affichées directement par la fonction `print_consttype` :

```
1  let print_consttype = function
2    Int_type ->  print_string "int"
3  | Float_type -> print_string "float"
4  | String_type -> print_string "string"
5  | Bool_type -> print_string "bool"
6  | Unit_type -> print_string "unit";;
```

Impressions des types et des erreurs (2)

La fonction var_name retourne une chaîne de caractères unique pour chaque entier passé en argument :

```
let string_of_char c =
  let s = " " in s.[0] <- c; s;;
let var_name n =
  let rec name_of n =
    let q,r = ((n / 26), (n mod 26)) in
      if q=0 then string_of_char(Char.chr (96+r))
      else (name_of q)^(string_of_char(Char.chr (96+r)))
  in
    "'"^(name_of n);;
```

La fonction principale est l'impression d'un schéma de type. Le seul problème est de trouver le nom d'une variable de types. la fonction locale names_of retourne les variables quantifiées, chacune associée à une chaîne de caractères particulière. Si une variable de type n'est pas donc cet ensemble une exception est déclenchée, car il ne peut avoir d'expression globale contenant des variables de type non quantifiées.

Impression des types et des erreurs (3)

```
let print_quantified_type (Forall (gv,t)) =
  let names =
    let rec names_of = function
      (n,[]) -> []
    | (n,(v1::lv)) -> (var_name n)::(names_of (n+1,lv))
    in (names_of (1,gv))
  in
    let var_names = List.combine (List.rev gv) names in
      let rec print_rec = function
        Var_type {contents=(Instanciated t)} -> print_rec t
      | Var_type {contents=(Unknown n)} ->
          let name =
            ( try List.assoc n var_names
              with Not_found ->
                raise (Failure "Non quantified variable in type"))
          in print_string name
      | Const_type ct -> print_consttype ct
      | Pair_type(t1,t2) -> print_string "("; print_rec t1;
          print_string " * ";  print_rec t2;  print_string ")"
      | List_type t ->
          print_string "(("; print_rec t; print_string ") list)"
      | Fun_type(t1,t2) -> print_string "("; print_rec t1;
          print_string " -> "; print_rec t2; print_string ")"
  in  print_rec t;;
```

Impression des types et des erreurs (4)

L'impression d'un type simple utilise l'impression des chémas de type en créant pour l'occasion une quantification sur toutes les variables libres du type.

```
1  let print_type t =
2    print_quantified_type (Forall(free_vars_of_type ([],t),t));;
```

Impression des types et des erreurs (5)

Il est fort intéressant d'afficher les causes d'un échec de typage, comme par exemple les types mis en cause lors d'une erreur de typage ou le nom d'une variable indéfinie. La fonction `typing_handler` reçoit en argument une fonction de typage, un environnement et une expression à typer. Elle capture les exceptions dues à une erreur de typage, affiche un message plus clair et déclenche une nouvelle exception.

```
1  let typing_handler typing_fun env expr =
2    reset_unknowns();
3    try typing_fun env expr
4    with
5      Type_error (Clash(lt1,lt2)) ->
6        print_string "Type clash between "; print_type lt1;
7        print_string " and "; print_type lt2;  print_newline();
8        failwith "type_check"
9    | Type_error (Unbound_var s) ->
10       print_string "Unbound variable ";
11       print_string s; print_newline();
12       failwith "type_check";;
```

Fonction principale

La fonction `type_check` type l'expression `e` par la fonction `type_expr` avec l'environnement de typage initial et affiche le type calculé de `e`.

```
let type_check e =
  let t =
    typing_handler type_expr initial_typing_env e in
    let qt =
      snd(List.hd(generalize_types initial_typing_env ["it",t])) in
        print_string "it : "; print_quantified_type qt; print_newline();;
```

Exemples

```
type_check (Const (Float 3.2));;
type_check (Abs ("x", Var "x"));;
type_check (Letin ("f", Abs ("x", Pair(Var "x", Var "x")),
                  ( Pair ((App (Var  "f", Const (Int 3))),
                          (App (Var "f", Const (Float 3.14)))))));;
type_check (App (Var "*", Pair (Const (Int 3), Const (Float 3.1))));;
type_check (Cond (
    App (Var "=", Pair(Const(Int 0), Const (Int 0))),
    Const(Int 2), Const(Int 5)));;
type_check (
  Letrecin ("fact",
    Abs ("x",
         Cond (App (Var "=",Pair(Var "x",Const(Int 1))),
               Const(Int 1),
               App (Var "*",
                    Pair(Var "x",
                         App (Var "fact",
                              App (Var "-",Pair(Var "x",Const(Int 1) ))
                             )
                        )
                   )
              )
        ),
    App (Var "fact", Const (Int 4))));;
```

Remarques sur le typage d'OCaml

Le typeur de mini-ML ne tenait pas compte des valeurs mutables (physiquement modifiables). OCaml est un langage fonctionnel muni d'effets de bord.
Si les primitives de constructions de valeurs mutables étaient traités comme des valeurs polymorphes classiques le *typeur* ne détecterait pas les inconsistances de types. Le programme suivant en est un exemple :

```
1  let x = ref (fun x -> x)
2  in
3    x:=(fun x -> x + 1);
4    !x true;;
```

et entraîne une erreur de typage.

Polymorphisme et valeurs mutables

- Une solution serait de ne construire que des valeurs mutables monomorphes.
- Comme cette contrainte est trop forte, il est possible de créer des valeurs mutables polymorphes spéciales, en utilisant des variables de types faibles pour celles-ci. L'idée est d'assurer lors de l'instanciation de cette variable de type, dans d'une application, que l'on obtient un type simple ou une variable de type liée au contexte de typage.
- Pour cela on différencie les expressions en deux classes : les expressions *non expansives* incluant principalement les variables, les constructeurs et l'abstraction et les expressions *expansives* incluant principalement l'application. Seules les expressions *expansives* peuvent engendrer une exception ou étendre le domaine (incohérence de types).

Expressions non-expansives (1)

Voici le texte de la fonction de reconnaissance d'expression non expansives du compilateur OCaml 3.0 (fichier `typecore.ml`) :

```
let rec is_nonexpansive exp =
  match exp.exp_desc with
    Texp_ident(_,_) -> true
  | Texp_constant _ -> true
  | Texp_let(rec_flag, pat_exp_list, body) ->
      List.for_all (fun (pat, exp) -> is_nonexpansive exp) pat_exp_list &
      is_nonexpansive body
  | Texp_apply(e, None::el) ->
      is_nonexpansive e &&
      List.for_all (function None -> true | Some exp -> is_nonexpansive e) el
  | Texp_function _ -> true
  | Texp_tuple el ->
      List.for_all is_nonexpansive el
  | Texp_construct(_, el) ->
      List.for_all is_nonexpansive el
```

Expressions non-expansives (2)

```
1   | Texp_variant(_, Some e) -> is_nonexpansive e
2   | Texp_variant(_, None) -> true
3   | Texp_record(lbl_exp_list, opt_init_exp) ->
4       List.for_all
5         (fun (lbl, exp) -> lbl.lbl_mut = Immutable & is_nonexpansive exp)
6         lbl_exp_list &&
7       (match opt_init_exp with None -> true | Some e -> is_nonexpansive e)
8   | Texp_field(exp, lbl) -> is_nonexpansive exp
9   | Texp_array [] -> true
10  | Texp_new (_, cl_decl) when Ctype.class_type_arity cl_decl.cty_type > 0 ->
11      true
12  | _ -> false ;;
```

Les expressions suivantes sont non-expansives (donc pourront être généralisées :

- les constantes, les identificateurs,
- les n-uplet si toutes les sous-expressions le sont, les constructeurs constants, les constructeurs d'arité 1 si ce constructeur n'est pas mutable et si l'expression passée au constructeur est elle aussi non-expansive,

Expressions non-expansives (3)

- les déclarations locales (let et let rec) si l'expression finale et toute les expressions des déclarations locales sont non-expansives,
- l'abstraction,
- la récupération d'exceptions si l'expression à calculer et toutes les expressions de partie droite du filtrage le sont, la séquence si le dernier élément de la séquence l'est,
- la conditionnelle si les branches *then* et *else* le sont, les expressions contraintes par un type si la sous-expression l'est,
- le vecteur vide,
- les enregistrements si tous ces champs sont non-mutables et si chaque expression associée à un champ l'est,
- l'accès à un champ d'un enregistrement si l'expression correspondante à l'enregistrement l'est, les filtrages de flots et les gardes si l'expression associée l'est.

Expressions non-expansives (4)

Toutes les autres expressions sont expansives.

En OCaml on pourra généraliser (c'est-à-dire construire un schéma de type) les variables de type rencontrées dans des expressions *non expansives* et ne pas généraliser (variable de type faible notée `_'a`) celles rencontrées dans des expressions *expansives*.

Ces variables de type faible pourront être instanciées ultérieurement dans le typage d'un module. Il est à noter que ces variables de type faible ne peuvent sortir d'un module (sans le risque de réemplois ultérieurs incompatibles).

Exemples (1)

Voici deux séries d'exemples de typage d'expressions non-expansives et d'expressions expansives :

avec effets de bord

OCaml 4.0	commentaires
expression non expansive	
`let nref x = ref x;;` `nref : 'a -> 'a ref = <fun>`	abstraction schéma de type
expression expansive	

`let x = ref [];;`
`x : '_a list ref = ref []`
`x:=[3];;`
`x;;`
`int list ref`
	ref est un constructeur mutable
	variable de type faible _a
	affinage du type de x

Exemples (2)

sans effets de bord

OCaml 4.0	commentaires
`let a x y = x y;;`	le combinateur A
`('a -> 'b) -> 'a -> 'b`	son schéma de type
`let g = a id;;`	une application partielle
`'_a -> '_a`	son type non généralisé
`g 3;;`	affinage du type de g
`int = 3`	
`g;;`	
`int -> int`	

Bibliographie

- R. Milner. “A theory of type polymorphism in programming”. J. Comput. Syst. Sci. 1978.
- X. Leroy. “Typage polymorphe d’un langage algorithmique”, Thèse de doctorat université Paris 7, 1992. décrit le typage polymorphe pour les effets de bords, les canaux de communications et les continuations dans un langage algorithmique fonctionnel.
- Wright (Andrew K.). “Polymorphism for Imperative Languages without Imperative Types”. Rapport technique n 93-200, Rice University, 1993. introduit la «value restriction» pour le typage des traits impératifs.
- Engel (Emmanuel). “Extensions sûres et praticables du système de types de ML en présence d’un langage de modules et de traits impératifs”. Thèse de doctorat université Paris 11, 1998. compare différents systèmes de typage polymorphe pour les effets de bords en présence de modules poru le langage Caml.
- Pierce (Benjamin). “Types and Programming Languages”. MIT Press, 2002.