Chapitre 2

Introduction au λ-calcul typé

2.1 Introduction au λ-calcul typé

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).

2.1.1 Les types simples pour le λ-calcul typé

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 $\to$) par :

– si $\alpha \in \mathcal{P}$ alors $\alpha \in \mathcal{T}$

– si $\sigma, \tau \in \mathcal{T}$ alors $\sigma \to \tau \in \mathcal{T}$.

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 $C = (x_1 : \sigma_1), \ldots, (x_n : \sigma_n)$.

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 : 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 $(f : \alpha \to \alpha) \vdash \lambda f.\lambda x.f(fx) : (\alpha \to \alpha) \to \alpha \to \alpha$

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

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

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

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).

2.1.2 Synthèse par unification

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 $\to$ des types). Les variables que nous considérerons sont celles de l'ensemble $\mathcal{P}$.

Rappelons qu'une substitution est une application $\mathcal{P} \to \mathcal{T}$ ($\mathcal{T}$ 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.

Définition 1 *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 1 *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.

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.

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

```
type oType =
  Tvar of string
| 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 :

```
(*  val substitute : (string * oType) list -> oType -> oType = <fun>↩
     *)

let substitute s t =
  let rec  sub_rec t = match t with
     (Tvar a) as alpha -> (try List.assoc a s with Not_found -> ↩
         alpha)
   | Arrow(oT1,oT2)    -> Arrow(sub_rec oT1, sub_rec oT2)
  in sub_rec t
;;
```

et voici la composition de deux substitutions :

```
(*  val compsubst :  (string * oType) list -> (string * oType) list ↩
    -> (string * oType) list =  <fun> *)

let compsubst s t =
  (List.map (function (a,oT) -> (a,substitute s oT)) t) @ s;;
```

La fonction `occur` teste l'existence d'une variable dans un type.

```
(* val occur : string -> oType -> bool = <fun> *)

let occur v t =
  let rec occ_rec t = match t with
    Tvar b -> b=v
  | Arrow(oT1,oT2) -> occ_rec oT1 or occ_rec oT2
  in occ_rec t
;;
```

La fonction `unify` calcule l'unificateur principal pour deux types simples. Elle retourne cet unificateur quand il existe ou échoue en déclenchant l'exception `Failure "unify"`.

```
(* val unify : oType * oType -> (string * oType) list = <fun> *)

let rec unify c = match c with
    (Tvar a), oT ->
      if oT=Tvar a then []
      else if occur a oT then raise (Failure "unify")
           else [a,oT]
```

```
8   | oT,Tvar a ->
9       if occur a oT then raise (Failure "unify")
10      else [a,oT]
11  | Arrow(oS1,oS2), Arrow(oT1,oT2) ->
12      let s=unify(oS1,oT1)  in
13          compsubst   (unify(substitute s oS2,
14                             substitute s oT2)) s
15 ;;
```

Les exemples suivants calculent l'unificateur principal de deux types. Cette substitution est ensuite appliquée aux types pour vérifier leur égalité après substitution.

1. Un des cas les plus simples où une variable de type est substituée par une autre :

```
# let oT1=(Tvar "x");;
val oT1 : oType = Tvar "x"
# let oT2=(Tvar "y");;
val oT2 : oType = Tvar "y"
# let s1 = unify (oT1,oT2);;
val s1 : (string * oType) list = [("x", Tvar "y")]
# substitute s1 oT1 = substitute s1 oT2;;
- : bool = true
```

2. Un cas classique où une variable est substituée par un type fonctionnel :

```
# let oT3 = (Arrow (Tvar "a", Arrow (Tvar "b",Tvar "c")));;
val oT3 : oType = Arrow (Tvar "a", Arrow (Tvar "b", Tvar "c"))
# let oT4 = (Arrow (Arrow (Tvar "d",Tvar "e"), (Arrow (Tvar "f"↩
    ,Tvar "g"))));;
val oT4 : oType =
  Arrow (Arrow (Tvar "d", Tvar "e"), Arrow (Tvar "f", Tvar "g")↩
      )
# let s2 = unify (oT3,oT4);;
val s2 : (string * oType) list =
  [("a", Arrow (Tvar "d", Tvar "e")); ("b", Tvar "f"); ("c", ↩
      Tvar "g")]
# substitute s2 oT3;;
- : oType = Arrow (Arrow (Tvar "d", Tvar "e"), Arrow (Tvar "f",↩
     Tvar "g"))
# substitute s2 oT4;;
- : oType = Arrow (Arrow (Tvar "d", Tvar "e"), Arrow (Tvar "f",↩
     Tvar "g"))
```

3. Un exemple où l'unification fait disparaitre presque toutes les variables :

```
# let oT5 = (Arrow (Arrow (Tvar "d",Tvar "a"), (Arrow (Tvar "a"↩
    ,Tvar "g"))));;
val oT5 : oType =
  Arrow (Arrow (Tvar "d", Tvar "a"), Arrow (Tvar "a", Tvar "g")↩
      )
# let oT6 = (Arrow (Arrow (Tvar "a",Tvar "e"), (Arrow (Tvar "a"↩
    ,Tvar "d"))));;
val oT6 : oType =
```

```
6   Arrow (Arrow (Tvar "a", Tvar "e"), Arrow (Tvar "a", Tvar "d")↩
        )
7  # let s3 = unify (oT5,oT6);;
8  val s3 : (string * oType) list =
9    [("d", Tvar "e"); ("a", Tvar "e"); ("g", Tvar "e")]
10 # substitute s3 oT5 = substitute s3 oT6;;
11 - : bool = true
```

4. Enfin un cas d'échec :

```
1  # unify (oT3,oT4);;
2  - : (string * oType) list =
3  [("a", Arrow (Tvar "d", Tvar "e")); ("b", Tvar "f"); ("c", Tvar↩
       "g")]
4  # unify (oT3,oT5);;
5  Exception: Failure "unify".
```

On peut maintenant donner l'algorithme de typage.

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).

Cette substitution rendue par l'algorithme s'applique aux variables de type du contexte, et contient les contraintes produites par le typage du terme.

- $T(x, C) = (C(x), id)$
- 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 $u \circ t \circ s$.

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 :

```
1 type term= Var of string
2      | Abs of string * term
3      | 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>
```

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

```
(*  val oTYPE : (string * oType) list -> term -> oType * (string * oType) list =  <fun> *)

let rec oTYPE oC t = match t with
     Var x    -> List.assoc x oC , []
  |  Abs(x,oM) ->
       let alpha=gensym "a" in
         let (sigma,s) = oTYPE ((x,Tvar alpha)::oC) oM in
           Arrow( (try List.assoc alpha s with _ ->  Tvar alpha),
                        sigma), s
  |  App(oM,oN) ->
       let (sigma,s)=oTYPE oC oM in
         let (tau,t) = oTYPE (List.map (function (x,phi) ->
                                 (x,substitute s phi)) oC) oN   in
           let alpha=gensym "a" in
             let u= unify(substitute t sigma, Arrow(tau,Tvar alpha)) in
               (try List.assoc alpha u with _ ->  Tvar alpha),
                        compsubst u (compsubst t s)
;;
```

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));;
```

```
8 |((a5 -> a6) -> (a5 -> a6))
9 |- : unit = ()
```

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 = ()
```

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".
```

2.1.3 Normalisation

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

On n'en donnera pas la preuve.

2.1.4 Polymorphisme

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 $\forall\alpha.\sigma \in \mathcal{S}$

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)

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

Ce qui nous permet de typer `let f =` $\lambda x.x$ `in ff` :

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

2.1.5 Exercices

1. Vérifier le type de S.
2. Écrire une fonction `new_print_type` qui n'affiche pas la représentation interne du type, comme celle indiquée ci-desssus, mais une représentation externe à la manière de Caml. Par exemple :

   ```
   new_print_type  (fst (TYPE [] A));;
   ```

   affichera `('a -> 'b) -> ('a -> 'b)`.
3. Typer : `let` deux `=` $\lambda f.\lambda x.f(fx)$ `in` deux deux

2.1.6 Bibliographie

– [**?**], R. Milner. "A theory of type polymorphism in programming" J. Comput. Syst. Sci. 1978

Premier article où le typage polymorphe de ML est décrit.

et les ouvrages cités dans la bibliographie sur le λ-calcul parlant du typage.