Typage et Analyse Statique

Cours 2

Emmanuel Chailloux

Parcours Science et Technologie du Logiciel
Master mention Informatique
Sorbonne Université

année 2023-2024

Plan du cours 2

Polymorphisme paramétrique:

- λ-calcul typé
  - types simples
    - vérification d'un type
    - unification
    - synthèse de types
  - schémas de types et polymorphisme

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 (ou* environnement*) 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 (1)

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).

On parle alors de jugement de typage, noté $C \vdash M : \tau$, qui indique que sous le contexte C, le terme M est typable avec le type τ.

Le parenthésage du constructeur de type fonctionnel ($\to$) est à droite : $\alpha \to \beta \to \alpha \equiv \alpha \to (\beta \to \alpha)$

typer un terme (2)

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)

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

- typer une variable : c'est vérifier qu'elle apparait dans le contexte.
- typer une abstraction : lui associer un type $\tau \to \sigma$ si le paramètre a un type τ et le corps un type σ.
- typer une application : M doit avoir un type fonctionnel et N le type du paramètre de M, et MN a le type du résultat.

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

Exemple de typage (2)

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

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

On a donné une méthode de *vérification* de typage (lecture de haut en bas), mais pas de *synthèse* (création de l'arbre de bas en haut) d'un type.

Synthèse (ou inférence) (1)

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 $f\ x$ il faut que σ soit de la forme $\tau \to \sigma_1$ et alors $f\ x$ est du type σ_1. Comme on a pu former $f(f\ x)$, 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).

L'idée est de poser les contraintes de typage en remontant la dérivation, puis de les résoudre par unification. Si échec le terme n'est pas typable.

Synthèse (ou inférence) (2)

En partant du terme à typer , on crée des contraintes de typage à chaque règle utilisée dans l'arbre de dérivation, puis on unifie ces contraintes. Si l'unification réussit on récupère le type le plus général, et sinon cela indique que le terme n'est pas typable.

avec $K = \lambda z_1 z_2.z_1$, $I_1 = \lambda y.y$ et $I_2 = \lambda x.x$, on cherche à typer $K\ I_1\ I_2$ on construit l'arbre de dérivation :

$$\dfrac{\dfrac{\dfrac{z_1 : t_4, z_2 : t_1 \vdash z1 : t_0}{z1 : t_4 \vdash \lambda z_2.z_1 : t_1 \to t_0}}{\vdash K : t_4 \to t_1 \to t_0} \qquad \dfrac{y : t_5 \vdash y : t_6}{\vdash I_1 : t_4}}{\vdash K\ I_1 : t_1 \to t_0} \qquad \dfrac{x : t_2 \vdash x : t_3}{\vdash I_2 : t_1}$$
$$\vdash K\ I_1\ I_2 : t_0$$

et on récupère cette liste de contraintes de typage :
$\{t_1 = t_2 \to t_3; t_2 = t_3; t_4 = t_5 \to t_6; t_5 = t_6; t_0 = t_4\}$
qui s'unifie en :
$\{t_0 = t_4 = \alpha \to \alpha; t_1 = \beta \to \beta; t_2 = t_3 = \alpha; t_5 = t_6 = \beta\}$

Synthèse (ou inférence) (3)

Termes non typables:

échec de l'unification sur le typage du terme $\lambda x.x\ x$:

- la variable x doit être d'un type α et en même temps d'un type $\alpha \rightarrow \beta$ car x s'applique sur x.

Synthèse (ou inférence) (4)

L'exploration du terme construit la dérivation avec :

- les règles **(Var)** produisent des contraintes sur les variables de type
- les règles **Abs** produisent des contraintes et engendre des variables
- les règles **App** engendrent des variables de type

On va s'intéresser à l'algorithme d'Hindley-Milner pour un λ-calcul simplement typé.

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".
```

Propriétés (1)

- subject reduction :
  Si $C \vdash M : \tau$ et $M \to M'$, alors $C \vdash M' : \tau$ (le type d'un calcul est le type de son résultat)
- normalisation forte :
  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.

Propriétés (2)

types habités:

On dit que le type τ est habité s'il existe M tel que $C \vdash M : \tau$

- $\alpha \to \alpha$ est habité par $\lambda x.x$
- $\alpha \to \beta$ n'est pas habité
- $(\alpha \to \beta) \to \alpha \to \beta$?
- $(\alpha \to \beta) \to \beta$?

Indication : utilisation des types des paramètres pour le type résultat.

thèse Gabriel Scherer : " Which types have a unique inhabitant ? Focusing on pure program equivalence." (2016)

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.

Polymorphisme (2)

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 (3)

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)

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

Polymorphisme (4)

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

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

C'est le *let* qui introduit le polymorphisme.