Vers un langage typé pour la programmation modulaire

UMONS

23 juin 2017

Introduction - OCaml

OCaml

Commençons par parler d'OCaml...

- langage fonctionnel : les fonctions sont des citoyens de première classe.
- statiquement typé : les types sont assignés à la compilation, non à l'exécution. $\neq$ dynamiquement typé (Python).
- dispose d'algorithmes d'inférence de type : pas obligé d'assigner manuellement un type à un terme (contrairement à C ou Java).
- divisé en deux « langages » :
  - langage de base : fonctions, enregistrements, tuples, variables, ...
  - langage de modules : modules (type = signature), foncteurs.
- **Problème :** les deux langages possèdent des algorithmes d'inférence de types différents et sont gérés différement par le compilateur.

But du mémoire

- Etudier un calcul théorique où les deux langages (et en particulier les règles de typage et de sous-typage) sont unifiés. Calcul choisi : DOT (2016).
- Représenter les modules et les enregistrements de la même manière.
- (Apport principal de ce mémoire). Implémenter un langage de surface, proche d'OCaml, un algorithme de typage et un algorithme de sous-typage pour écrire des programmes DOT. Langage : RML

Programmation fonctionnelle et typage

λ-calcul non typé

Calcul = syntaxe (termes) + sémantique (règles d'évaluation, non discutée).
Soit V un ensemble infini dénombrable de variables. La syntaxe du λ-calcul est définie comme le plus petit ensemble Λ tel que

1. $V \subseteq \Lambda$
2. $\forall u, v \in \Lambda, u\, v \in \Lambda$
3. $\forall x \in V, \forall u \in \Lambda, \lambda x.\, u \in \Lambda$

$\Leftrightarrow$

$t ::=$		terme
	x	var
	$t\, t$	app
	$\lambda x.\, t$	abs

- $u\, v \rightarrow$ application d'une fonction u à un paramètre v.
- $\lambda x.\, u \rightarrow$ définition d'une fonction (anonyme) qui prend un paramètre x et retourne u.

Typage

- Classer les termes en fonction de leur nature $\rightarrow$ notion de type.
- Exemple avec le terme $u\,v$: u doit être une fonction qui prend un paramètre v d'un certain type T_1 et retourne un terme d'un type T_2. On note $T_1 \rightarrow T_2$ le type de u (type flèche).
- Formellement et de manière générale, on se donne τ, un ensemble de types, les éléments étant notés T, et on définit une **relation de typage** entre Λ et τ. On note $t : T$ pour dire que t est de type T. On dit aussi que t est bien typé.

λ-calcul simplement typé - Syntaxe des types

Soit $\mathcal{B}$ un ensemble de types de base, ses éléments étant notés B.

$$
\begin{array}{llr}
T ::= & & \text{types} \\
& B & \text{base} \\
& T \to T & \text{type des fonctions}
\end{array}
$$

Contexte et jugement de typage, différence de syntaxe

- $\lambda x.\, t$: type de x ? $\Rightarrow$ on annote la variable avec son type. $\lambda x.\, t$ devient $\lambda x : T.\, t$.
- un terme t peut contenir des variables $\Rightarrow$ pour typer t, besoin de connaître le type de celles-ci $\Rightarrow$ **contexte ou environnement de typage** $\Gamma =$ suite (x_i, T_i) où x_i est une variable et T_i est le type de x_i.
- $t : T$ devient $\Gamma \vdash t : T$ (appelé **jugement de typage**).

λ-calcul simplement typé - Règles de typage

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 .\, t : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash u : T_1 \to T_2 \qquad \Gamma \vdash v : T_1}{\Gamma \vdash u\, v : T_2} \quad \text{(T-APP)}$$

λ-calcul simplement typé - Arbre de dérivation

Une succession d'utilisations de règles de typage mène à **un arbre de dérivation**.

$$\dfrac{\dfrac{x : T_1 \to T_2 \in \Gamma}{\Gamma \vdash x : T_1 \to T_2}\,(\text{T-VAR}) \qquad \dfrac{y : T_1 \in \Gamma}{\Gamma \vdash y : T_1}\,(\text{T-VAR})}{\dfrac{\Gamma, x : T_1 \to T_2, y : T_1 \vdash x\,y : T_2}{\dfrac{\Gamma, x : T_1 \to T_2 \vdash \lambda y : T_1 .\, x\,y : T_1 \to T_2}{\Gamma \vdash \lambda x : T_1 \to T_2 .\, \lambda y : T_1 .\, x\,y : (T_1 \to T_2) \to T_1 \to T_2}\,(\text{T-ABS})}\,(\text{T-ABS})}\,(\text{T-APP})$$

Pour le λ-calcul simplement typé : il existe au plus un arbre de dérivation.
Pas toujours le cas (exemple : DOT).

λ-calcul simplement typé - Sûreté

Théorèmes importants :

1. Préservation : si $t : T$ et $t \to t'$, alors $t' : T$.
2. Progression : si $t : T$, alors soit t est une valeur, soit $t \to t'$.

Preuves : lemmes intermédiaires + induction sur l'arbre de dérivation.

Sous-typage

- But : affiner notre relation de typage en définissant une relation binaire (dite de **sous-typage**) sur les types.
- Notation : $S <: T$ (S est sous-type de T).
- Ajout d'une règle de typage qui crée un lien entre la relation de typage et la relation de sous-typage.

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

- La relation est souvent transitive et réflexive :

$$\frac{S <: T \qquad T <: U}{S <: U} \quad \text{(S-TRANS)} \qquad\qquad T <: T \quad \text{(S-REFL)}$$

Enregistrement

- enregistrement = ensemble de labels liés à des termes. Par exemple $\{l_1 = 5; l_2 = \lambda x.\, 42\}$.
- Les champs ne sont pas mutuellement dépendants !
- Formellement, la syntaxe des termes et des types deviennent

$$
\begin{array}{llr}
t ::= & & \text{terme} \\
 & x & \text{var} \\
 & t\ t & \text{app} \\
 & \lambda x : T.\, t & \text{abs} \\
 & \{l_i = t_i\}^{1 \leq i \leq n} & \text{enreg} \\
 & t.l & \text{proj}
\end{array}
\qquad
\begin{array}{llr}
T ::= & & \text{type} \\
 & T \to T & \text{fonction} \\
 & \{l_i : T_i\}^{1 \leq i \leq n} & \text{enreg}
\end{array}
$$

DOT

DOT - Description

- Ajoute les types à l'intérieur des enregistrements $\Rightarrow$ types dépendants $(x.A)$.
- Un enregistrement peut avoir des champs mutuellement dépendants grâce à une variable interne.
- Les fonctions deviennent dépendantes : le types de retour peut dépendre du paramètre.

DOT - Syntaxe des termes

$t ::=$	terme
x, y	var
$\lambda x : T.\, t$	abs
$x\, y$	app
$let\ x \ = \ t\ in\ t$	let
$\nu(x : T^x)d$	rec
$x.a$	champ proj

$d ::=$	decl
$\{a = t\}$	champ
$\{A = T\}$	type
$d \wedge d$	aggregation

DOT - Syntaxe des types

$S, T ::=$		type
	Top	top
	$Bottom$	bottom
	$\forall(x : S)T^x$	fonction
	$\{A : S..T\}$	type decl
	$\{a : T\}$	champ decl
	$x.A$	type proj
	$\mu(x : T^x)$	rec
	$S \wedge T$	inter

DOT : quelques règles de typage

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \quad (<:\mathrm{SEL}) \qquad \frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \quad (\mathrm{SEL}<:)$$

$$\frac{\Gamma \vdash x : T^x}{\Gamma \vdash x : \mu(z : T^z)} \quad (\mathrm{VAR\text{-}PACK}) \qquad \frac{\Gamma \vdash x : \mu(z : T^z)}{\Gamma \vdash x : T^x} \quad (\mathrm{VAR\text{-}UNPACK})$$

DOT - Remarques

- Pas de règles de sous-typage pour les types récursifs $\Rightarrow$ utilisation de règles de typage (VAR-PACK, VAR-UNPACK) suivies de règles de sous-typage pour les comparer.
- Règles de sous-typage dépendent du typage (SEL $<:$ et $<:$ SEL). **Pas usuel**.

Implémentation de DOT en OCaml : RML

Implémentation : remarques

1. Aucun article ne décrit l'implémentation d'algorithmes de typage et de sous-typage pour DOT.
2. Besoin de définir un langage de surface (= un langage plus haut niveau et plus facile à lire que le calcul théorique).
3. RML implémente des sucres syntaxiques pour écrire plus facilement des programmes (fonction à plusieurs arguments, applications avec plusieurs arguments).
4. RML implémente un algorithme d'inférence de type pour les modules (terme $\nu(x)d$ en plus de $\nu(x : T)d$). Dans DOT, la définition d'un enregistrement récursif est obligatoirement liée à un type.

RML - module = enregistrement

```
let module Point2D = struct(point)
  type t = { x : Int.t ; y : Int.t }
  let add = fun(p1 : point.t, p2 : point.t) ->
    { x = Int.plus p1.x p2.x ; y = Int.plus p1.y p2.y }
end;;
```

- $\nu(x)d \rightarrow$ struct d end
- $\mu('self : T) \rightarrow$ enregistrement
- $\{a = t\} \rightarrow$ let a = t
- $\{A = T\} \rightarrow$ type a = t
- $\lambda x : T_1.\, t \rightarrow$ fun(x : T_1) $\rightarrow$ t

RML - module = enregistrement - Type

```
sig(point)
  type t = sig('self) (* enregistrement *)
    val x : Int.t
    val y : Int.t
  end .. sig('self)
    val x : Int.t
    val y : Int.t
  end
  val add : forall(p1 : point.t) (* fonction *)
            forall(p2 : point.t)
    sig('self)
      val x : Int.t
      val y : Int.t
    end
end
```

RML - foncteur = fonction

```
let module MakePoint2D =
  fun(typ : sig(self)
    type t
    val plus : self.t -> self.t -> self.t
  end) ->
  struct(point)
    type t = { x : typ.t ; y : typ.t }
    let add = fun(p1 : point.t, p2 : point.t) ->
      let x' = typ.plus p1.x p2.x in
      let y' = typ.plus p1.y p2.y in
      { x = x' ; y = y' }
end;;
```

RML - foncteur = fonction - Type

```
forall(typ : sig(self)
  type t = Nothing .. Any
  val plus : self.t -> self.t -> self.t
end) sig(point)
  type t = sig('self)
    val x : typ.t
    val y : typ.t
  end .. sig('self)
    val x : typ.t
    val y : typ.t
  end
  val add : forall(p1 : point.t)
            forall(p2 : point.t)
    sig('self)
      val x : typ.t
```

Implémentation : pas si facile...

Passer de la théorie à l'implémentation n'est pas facile !

1. Gestion des variables libres et liées (utilisation d'AlphaLib).
2. Un même jugement de typage peut être dérivé de plusieurs manières.
3. Une même question $S <: T$ peut être posée plusieurs fois (ex : liste avec tail).
4. Indécidabilité du sous-typage $\Rightarrow$ il n'existe pas d'algorithme qui termine sur chaque entrée et qui soit correct et complet.

Implémentation : problèmes

1. DOT n'est pas stable par insertion de binding locaux variable-variable.
2. (SEL < :) et (< : SEL) nécessite un algorithme intermédiaire (best_bound) dans le cas où une question $x.A <: y.A$ est posée. Ce dernier n'est pas évident !

$$\frac{\Gamma \vdash x : \{A : L..U\}}{\Gamma \vdash x.A <: U} \quad (\text{SEL-}<:)$$

devient

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash T <: \{A : L..U\} \qquad \Gamma \vdash U <: U'}{\Gamma \vdash x.A <: U'} \quad (\text{UN-SEL-}<:)$$

Travail futur

- Résoudre le problème des questions déjà posées (en cours, plusieurs idées).
- Evaluateur.
- Interpréteur intéractif.
- TODO