Vers un langage typé pour la programmation modulaire

UMONS

23 juin 2017

Introduction - OCaml

OCaml

Commençons par parler d'OCaml...

- langage fonctionnel : fonctions sont des citoyens de première classe et ce « nativement ».
- 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, ...
  - langage de modules : modules (type = signature), foncteurs.
- les deux langages possèdent des algorithmes d'inférence de types différents.

```
module Point2D = struct
  type t = { x : int ; y : int }
  let add = fun p1 -> fun p2 ->
    let x' = p1.x + p2.x in
    let y' = p1.y + p2.y in
    { x = x' ; y = y'}
end;;
(* Signature (type) de Point2D
module Point2D : sig
  type t = { x : int; y : int; }
  val add : t -> t -> t
end
*)
```

```
module MakePoint2D
    (T : sig type t val add : t -> t -> t end) =
  struct
    type t = { x : T.t ; y : T.t }
    let add = fun p1 -> fun p2 ->
      let x' = T.add p1.x p2.x in
      let y' = T.add p1.y p2.y in
      { x = x' ; y = y'}
  end;;
(* Signature de MakePoint2D
module MakePoint2D :
  functor (T : sig type t val add : t -> t -> t end) ->
    sig type t = { x : T.t; y : T.t; }
        val add : t -> t -> t
    end
*)
```

But du mémoire

- Définir un calcul théorique où les deux langages (et en particulier les règles de typage et de sous-typage) sont unifiés. Calcul choisi : DOT (2016).
- Unification de la notion de module et d'enregistrement. Ces deux termes et leurs types sont représentés de la même manière ($\neq$ en OCaml).
- 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 (apport principal de ce mémoire).

Programmation fonctionnelle et typage

λ-calcul non typé

Calcul = syntaxe (termes) + sémantique (règles d'évaluation).
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. $\forall u, v \in \Lambda, uv \in \Lambda$
3. $\forall x \in V, \forall u \in \Lambda, \lambda x.\, u \in \Lambda$

$\Leftrightarrow$

$$
\begin{array}{lll}
t ::= & & \text{terme} \\
 & x & \text{var} \\
 & t\,t & \text{app} \\
 & \lambda x.\, t & \text{abs}
\end{array}
$$

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

λ-calcul simplement typé

But : classifier les termes en fonction de leur nature.

λ-calcul simplement typé

Théoèmes importants qui permet d'affirmer « un terme bien typé ne bloque pas ».

1. Préservation : si un terme t est de type T et t s'évalue en t', alors t' est de type T.
2. Progression : si un terme t est de type T, alors soit t est une valeur, soit t s'évalue en t'.

Ces théorèmes sont vrais pour tous les calculs définis dans ces transparents.

(Polymorphisme par) sous-typage et enregistrement

Polymorphisme paramétré : Système F

But :

Système $F_{<:}$

But : unifier le polymorphisme par sous-typage et le polyphormisme paramétré.

Modules de première classe : DOT

Implémentation : RML

Problèmes :

1. Un même jugement de typage peut être dérivé de plusieurs manières.
2. Une même question peut être posée plusieurs fois.
3. Indécidabilité du sous-typage.