Typage Recursif

Sylvain Ribstein

Typage, 28 mars 2017

Master 2 de Recherche en Informatique
Paris 7 - Diderot

LangAST

```
type exp =
  | Var   of var
  | Const of const
  | Pair  of exp * exp
  | App   of exp * exp
  | Abs   of var * exp
  | Let   of var * exp * exp

and const =
  | Fct  of string
  | Bool of bool
  | Int  of int

and var = string

and t = exp
```

TypeAST

```
type ty =
  | TInt | TBool
  | Ground of ground
  | Cross of ty * ty
  | Arrow of ty * ty
  | Rec of ground * ty

and ground = string

and t = ty
```

Generation de Contrainte

Parcours en profondeur de l'expression :

- environnement des variables introduites
- type expecté de l'élément courant

Generation de Contrainte

```
let system_equation p =
  let rec aux env sys_eq expected= function
    | Var x ->
      let x_t =
        try (List.assoc x env)
        with _ -> raise (Unbound x)
      in
      (x_t, expected)::sys_eq
    | Const (Fct f)  -> (List.assoc f env, expected)::sys_eq
    | Const (Bool _) -> (TBool, expected)::sys_eq
    | Const (Int _)  -> (TInt,  expected)::sys_eq
    | Pair (m, n)    ->
      let m_t = fresh_ground () in
      let n_t = fresh_ground () in
      let sys_eq = aux env sys_eq m_t m in
      let sys_eq = aux env sys_eq n_t n in
      (Cross(m_t, n_t),expected)::sys_eq
```

Generation de Contrainte

```
    | App(func, arg) ->
      let arg_t = fresh_ground () in
      let func_t = Arrow(arg_t,expected) in
      let sys_eq = aux env sys_eq func_t func in
      aux env sys_eq arg_t arg
    | Abs(x, m)      ->
      let x_t    = fresh_ground ()  in
      let m_t    = fresh_ground ()  in
      let env    = (x, x_t)::env        in
      let sys_eq = aux env sys_eq m_t m in
      (Arrow(x_t, m_t), expected)::sys_eq
    | Let (x, m, n)  ->
      let x_t    = fresh_ground ()  in
      let sys_eq = aux env sys_eq x_t m in
      let env    = (x, x_t)::env        in
      aux env sys_eq expected n
in
let final_type = fresh_ground () in
final_type, (aux fct_type [] final_type p)
```

MGU avec type recursif

Pour chaque équation de contrainte généré :

- les règles de transformation sont appliquées
- les substitutions introduites sont appliquées au restes des équations

L'ensemble des substitutions sont renvoyées.

type recursif

- Si occurs-check alors introduction d'un type récursif
- Si un type recursif alors le type est déplié et l'unification reprend

MGU avec type recursif

```
let rec mgu_one = function
  | (t1, t2) when t1 = t2 -> []
  | (Ground x, Ground y) ->  [(x, Ground y)]
  | (TBool, Ground x) | (Ground x, TBool) -> [(x, TBool)]
  | (TInt, Ground x) | (Ground x, TInt ) -> [(x, TInt)]
  | (Arrow (x, y), Arrow (x', y')) -> mgu [(x, x');(y, y')]
  | (Cross (x, y), Cross (x', y')) -> mgu [(x, x');(y, y')]
  | (Ground x, t) | (t, Ground x) ->
    if (Type.occurs x t) then [(x, Rec (x, t))]
    else [(x, t)]
  | Rec (x, x_t), t
  | t, Rec (x, x_t) ->
    let x_t = apply [(x, Rec (x, x_t))] x_t in
    mgu_one (x_t, t)
  | t1, t2 -> raise (NoTypable (t1, t2))

and mgu = function
  | [] -> []
  | (x, y)::t ->
    let sub_tl = mgu t in
    let sub_hd = mgu_one ((apply sub_tl x), (apply sub_tl y)) in
    sub_hd @ sub_tl
```