unicaml

Type inference for a functional language

Implement the type inference algorithm presented in the given lecture.

Syntax of expressions and types

```
type ide = Ide of string;;

type exp =
    N of int
  | Val of ide
  | Add of exp * exp
  | Sub of exp * exp
  | Mul of exp * exp
  | Div of exp * exp
  | True
  | False
  | Eq of exp * exp
  | Leq of exp * exp
  | Not of exp
  | And of exp * exp
  | Or of exp * exp
  | If of exp * exp * exp
  | Let of ide * exp * exp
  | Letrec of ide * exp * exp
  | Fun of ide * exp
  | Apply of exp * exp
;;

type etype =
    TBool
  | TInt
  | TVar of string
  | TFun of etype * etype;;
```

Type constraints

The first phase of type inference consists in constructing a set of type constraints. This is the goal of the function `tconstraints`. The first argument is the expression to be analysed. The second argument is a type environment, with type `(ide * etype) list`. The result is a value of type `etype * (etype * etype) list`, where the first item is the type of the expression, and the second item is the list of constraints.

```
let rec tconstraints e tr = match e with
  N n -> (TInt,[])
| Val x ->  (applyenv tr x,[])
| Add (e1,e2)
| Sub (e1,e2)
| Mul (e1,e2)
| Div (e1,e2) ->
    let (t1,c1) = tconstraints e1 tr in
    let (t2,c2) = tconstraints e2 tr in
    let c = [(t1,TInt); (t2,TInt)] in
    (TInt, c @ c1 @ c2)
| True
| False -> (TBool,[])
| Eq (e1,e2)
```

```
  | Leq (e1,e2) ->
      let (t1,c1) = tconstraints e1 tr in
      let (t2,c2) = tconstraints e2 tr in
      let c = [(t1,TInt); (t2,TInt)] in
      (TBool, c @ c1 @ c2)
  | Not e1 ->
      let (t1,c1) = tconstraints e1 tr in
      let c = [(t1,TBool)] in
      (TBool, c @ c1)
  | And (e1,e2)
  | Or (e1,e2) ->
      let (t1,c1) = tconstraints e1 tr in
      let (t2,c2) = tconstraints e2 tr in
      let c = [(t1,TBool); (t2,TBool)] in
      (TBool, c @ c1 @ c2)
  | If(e0,e1,e2) ->
      let (t0,c0) = tconstraints e0 tr in
      let (t1,c1) = tconstraints e1 tr in
      let (t2,c2) = tconstraints e2 tr in
      let c = [(t0,TBool); (t1,t2)] in
      (t1, c @ c0 @ c1 @ c2)
  | Let (x,e1,e2) ->
      let (t1,c1) = tconstraints e1 tr in
      let (t2,c2) = tconstraints e2 (bind tr x t1) in
      (t2, c1 @ c2)
  | Letrec (x,e1,e2) ->
      let tx = gentide() in
      let (t1,c1) = tconstraints e1 (bind tr x tx) in
      let (t2,c2) = tconstraints e2 (bind tr x tx) in
      let c = [(tx,t1)] in
      (t2, c @ c1 @ c2)
  | Fun (x,e1) ->
      let tx = gentide() in
      let (t1,c1) = tconstraints e1 (bind tr x tx) in
      (TFun (tx,t1), c1)
  | Apply (e1,e2) ->
      let tx = gentide() in
      let (t1,c1) = tconstraints e1 tr in
      let (t2,c2) = tconstraints e2 tr in
      let c = [(t1,TFun(t2,tx))] in
      (tx, c @ c1 @ c2)
;;
```

```
val tconstraints : exp -> (ide * etype) list -> etype * (etype * etype) list = <fun>
```

The function gentide given below genereates a fresh type identifier.

```
let nextsym = ref (-1);;
let gentide = fun () -> nextsym := !nextsym + 1; TVar ("?T" ^ string_of_int (!nextsym));;
```

Unification

The second phase provides for finding a solution to the set of constraints constructed in the first phase. This is done by the `unify` function. The first argument is the list of constraints. The result, of type `(etype * etype) list`, is a list of substitution from type variables to types.

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

Some auxiliary functions are helpful.

- `applysubst1` substitutes a type for a type identifier in a type.
- `applysubst` substitutes a type for a type identifier in a list of constraints.
- `occurs` tells wheter a type identifier occurs in a type.

```
val applysubst1 : etype -> string -> etype -> etype = <fun>
val applysubst : (etype * etype) list -> string -> etype -> (etype * etype) list = <fun>
val occurs : string -> etype -> bool = <fun>
```

```
let rec applysubst1 t0 x t = match t0 with
    TInt -> TInt
  | TBool -> TBool
```

```
  | TVar y -> if y=x then t else TVar y
  | TFun (t1,t2) -> TFun (applysubst1 t1 x t, applysubst1 t2 x t)
;;

let rec applysubst l x t = match l with
    [] -> []
  | (t1,t2)::l' -> (applysubst1 t1 x t, applysubst1 t2 x t)::(applysubst l' x t)
;;

let rec occurs x t = match t with
  TInt
| TBool -> false
| TVar y -> x=y
| TFun (t1,t2) -> (occurs x t1) || (occurs x t2)
;;

let rec unify  l = match l with
  [] -> []
| (TInt,TInt)::l' -> unify l'
| (TBool,TBool)::l' -> unify l'
| (TVar x, t)::l' ->
    if occurs x t then failwith "Occurs check"
    else (TVar x,t)::(unify (applysubst l' x t))
| (t, TVar x)::l' ->
    if occurs x t then failwith "Occurs check"
    else (TVar x,t)::(unify (applysubst l' x t))
| (TFun(t1,t2),TFun(t1',t2'))::l' ->
    unify ((t1,t1') :: (t2,t2') :: l')
| _ -> failwith "Unsolvable constraints"
;;
```

Finally, the function `type_inference` combines the two phases above to infer a type for an expression, or return an error if the expression is not typable.

```
let type_inference e =
  let rec resolve t s = (match s with
    [] -> t
  | (TVar x, t')::s' -> resolve (applysubst1 t x t') s'
  | _ -> failwith ("Ill-formed substitution")) in
  let (t,c) = tconstraints e emptyenv in
  resolve t (unify c)
;;
```

```
val type_inference : exp -> etype = <fun>
```

Examples

```
let e0 = Let(Ide "succ",
             Fun(Ide "x", Add(Val(Ide "x"), N 1)),
             Apply(Val(Ide "succ"),N 8));;

let (t0,c0) = tconstraints e0 emptyenv;;
# val t0 : etype = TVar "?T8"
# val c0 : (etype * etype) list =
  [(TVar "?T7", TInt); (TInt, TInt);
   (TFun (TVar "?T7", TInt), TFun (TInt, TVar "?T8"))]

unify c0;;
# - : (etype * etype) list = [(TVar "?T7", TInt); (TVar "?T8", TInt)]

type_inference e0;;
# - : etype = TInt

let e1 = Letrec(Ide "fact",
                Fun(Ide "x",
                    If(Eq(Val(Ide "x"),N 0),
                       N 1,
                       Mul(Val(Ide "x"),
                           Apply(Val(Ide "fact"),Sub(Val(Ide "x"), N 1))))),
               Apply(Val(Ide "fact"),N 5));;

type_inference e1;;
# - : etype = TInt
```

```
let mul2 = Let(Ide "mul",
                 Fun(Ide "x", Fun (Ide "y", Mul(Val (Ide "x"), Val (Ide "y")))),
                 Fun(Ide "x", Apply(Apply (Val (Ide "mul"), N 2), Val(Ide "x"))))
;;

type_inference mul2;;
# - : etype = TFun (TInt, TInt)

let double = Fun(Ide "f",
                 Fun (Ide "x", Apply(Val(Ide "f"),Apply(Val(Ide "f"),Val(Ide "x")))))
;;

type_inference double;;
# - : etype = TFun (TFun (TVar "?T21", TVar "?T21"), TFun (TVar "?T21", TVar "?T21"))

let doubleboolnat =
    Let(Ide "double",
        Fun(Ide "f",
            Fun (Ide "x", Apply(Val(Ide "f"),Apply(Val(Ide "f"),Val(Ide "x"))))),
        Let(Ide "a", Apply(Apply(Val(Ide "double"),Fun(Ide "x", Apply(succ,Apply(succ,Val(Ide "x"))))),N 2),
            Let(Ide "b", Apply(Apply(Val(Ide "double"), Fun(Ide "x", Val(Ide "x"))), True),
                N 2)))
;;

type_inference doubleboolnat;;
# Exception: Failure "Unsolvable constraints".
```

hofl_type_inference.txt · Last modified: 2015/10/08 15:20 (external edit)