Project

The second part of your project for Programmation 1.2 is to complete your implementation of polymorphic type inference in order to handle references.

Recall that the project is due on **January 11, 2013**, after the last lab session. Oral presentations on both projects, Programmation 1.1 and 1.2, will take place on January 23, 2013.

You will find at `http://www.lsv.ens-cachan.fr/~schmitz/teach/2012_prog/part2/project/project.tar.gz` the files for this last part.

Deliverables. After scaffolding the project files, you will need to copy your files 'dag.ml' and 'unionfind.ml' from TP 2 into the 'fastunif' subdirectory. You should also copy your file 'typedtree.ml' from TP 4 into the root directory. You should then only modify the 'typedtree.ml' file, which ought to provide the two functions type_expr and print_exp_type defined in 'typedtree.mli'. Your project deliverables are

1. the three files 'dag.ml', 'unionfind.ml', and 'typedtree.ml', along with
2. a short report (of roughly two pages length) containing:
   - a general description of the datatypes used by your algorithm,
   - how your algorithm works,
   - how you tested your program,
   - which difficulties you encountered,
   - which extensions you implemented, and
   - anything else we should know.

The Language

Abstract Syntax. Here is the abstract syntax of expressions for this last part:

$$\begin{aligned} e \ ::= \ & c \mid x \mid \textbf{fun}\, x \to e \mid e\, e \mid \textbf{let}\, x = e\, \textbf{in}\, e \\ & \mid e + e \mid e - e \mid e * e \mid -e \mid e = e \mid e < e \mid e > e \\ & \mid e\, \textbf{or}\, e \mid e \parallel e \mid e \,\&\, e \mid e \,\&\&\, e \\ & \mid \textbf{let rec}\, x = e\, \textbf{in}\, e \mid e; e \mid \textbf{ignore}\, e \\ & \mid \textbf{ref} e \mid !e \mid e := e \end{aligned}$$

where c ranges over integer constants and x over identifiers.

Concrete Syntax. The project files contain a simplified OCaml parser, taken from the actual OCaml source distribution, in the 'ocaml' subdirectory. The abstract syntax trees it constructs are defined in 'parsetree.mli' located at the project root; you can pretty-print parse trees with the command 'viewast'—after running 'make'.

There is a slight incongruity in the proposed parse trees: the constructor for applications $e_1\, e_2$, namely 'Pexp_apply **of** expression $*$ expression', encodes applications in the opposite direction: the second child is applied with the first child as argument.

Types for Built-ins. The types for the new built-in functions in the language are the following:

```
val ref: 'a → 'a ref
val := : 'a ref → 'a → unit
val ! : 'a ref → 'a
val ignore: 'a → unit
```

Test Cases. The file 'tests.**in**' contains a few test cases; here are the outputs I obtain on the new tests:

```
# (* Theta handling *)
fun f → f (let f = fun x y → x in f 1);;
- : (('a → int) → 'b) → 'b
# fun f → (let f = fun x y → x in f 1) f;;
- : 'a → int
# (* let rec *)
let rec f = fun x → x + (f x) in f;;
- : int → int
# (* sequences *)
let id = fun x → x in
  let id' = id id in
  ignore (id' (0 < 1));
  id' 3;;
File "", line 48, characters 2-7:
Error: untypable
# let id = fun x → x in
  let id' = fun x → id (id x) in
  ignore (id' (0 < 1));
  id' 3;;
- : int
# (* ref *)
(ref);;
- : 'a → 'a ref
# (:=);;
- : 'a ref → 'a → unit
# let x = ref 0 in
  ignore (!x < 1);
  x := 1;
  !x - 1;;
```

```
- : int
# let f = fun x -> ref x in
  ignore (!f + 1);
  f := (0 < 1);;
File "", line 66, characters 10-12:
Error: untypable
# let f = ref (fun x -> x) in
  f := (fun x -> x + 1);
  (!f) 0;;
- : int
# let f = ref (fun x -> x) in
  f := (fun x -> x + 1);
  (!f) (0 < 1);;
File "", line 75, characters 2-14:
Error: untypable
```

Value Restricted Terms. In order to safely infer types in the presence of references and side-effects, a commonly used solution is to apply generalizations in '**let**' constructs to 'value restricted' terms only. You can look up on the Internet what that means.

This idea is used in OCaml. The first example with sequences

```
let id = fun x -> x in
  let id' = id id in
  ignore (id' (0 < 1));
  id' 3;;
```

is untypable due to this typing strategy; it could be accepted in other typing algorithms.

Extensions. You are free to extend this language. You should make conservative extensions, such that the types are not changed by your modifications.