Polymorphic Type Inference

The first part of your project for Programmation 1.2 is to implement the type inference algorithm seen in class, called '$PT(M, \Theta)$' in the lecture notes, on a small subset of the OCaml language. The second part of this project will add a few more constructs to the language. The project will be 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/tp4_typing/typing.tar.gz` the files for this first 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 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.

A Tiny Language

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

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

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 built-in functions in the language are the following:

```
val +   : int → int → int
val -   : int → int → int
val *   : int → int → int
val -u  : int → int          (* Unary 'minus' *)
val =   : 'a → 'a → bool
val <   : 'a → 'a → bool
val >   : 'a → 'a → bool
val or  : bool → bool → bool
val ||  : bool → bool → bool
val &   : bool → bool → bool
val &&  : bool → bool → bool
```

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

```
# 1;;
- : int
# (+);;
- : int → int → int
# (=);;
- : 'a → 'a → bool
# fun x → x;;
- : 'a → 'a
# 1 = 2;;
- : bool
# fun x y z t → (x = y) & (z = 1);;
- : 'a → 'a → int → 'b → bool
# let id = fun x → x in id x + 1;;
- : int
# let id = fun x → x in fun x y z → (id (x > (id 1)) & y);;
- : int → bool → 'a → bool
# fun y → let f = fun x → y in f 3;;
- : 'a → 'a
# let f = fun x y → x > 1 & y in f;;
- : int → bool → bool
# let f = fun x y → x > 1 & y in f 0 (1 = 1);;
- : bool
# let pair = fun a b f → f a b in pair 10 20;;
```

```
- : (int → int → 'a) → 'a
# let pair = fun a b f → f a b in
  let id = fun x → x in
  pair id id;;
- : (('a → 'a) → ('b → 'b) → 'c) → 'c
# let pair = fun a b f → f a b in
  let fst = fun p → p (fun x y → x) in
  fst (pair 10 12);;
- : int
# let id = fun x → x in let f = fun x → id 20 in id;;
- : 'a → 'a
```

What's Next? Next week: the second part will add lists, references, if-then-else, etc.