Project

1 Introduction

The second project is the design and implementation of type inference for a moral subset of ML. The deadline for this project is at

23:59 (local time) on January 5th, 2010

which is right after the last TP (so that you have another chance to ask questions and request clarifications).

The defense of the two projects will be on January 11th, 2010. You will have half an hour to present your projects and answer questions.

1.1 Deliverables

From the home page (`http://www.lsv.ens-cachan.fr/~ciobaca/p12.html`) of the TP you can download a scaffold for your project.

The scaffold consists of a lexer (ocamllex file lexer.mll) and a parser (ocamlyacc file parser.mly) which convert input into an abstract data type mimicking the abstract syntax of the subset of ML described in Section 2. Also included (in the file main.ml, function eval) is a toy interpretor (β-quality) which you can use to try out some examples.

You should modify the typing function (in the file main.ml) so that it outputs the type ~~or type schema~~ of every expression surrounded by the TYPE modifier. See Section 3 for details regarding the types that you are expected to handle. Feel free to place your code in a different file for better modularization.

You should not modify the files lexer.mll, parser.mly, mlsyntax.ml except if you plan to implement extensions. In this case, see Section 4.

Before the deadline, you are expected to hand over electronically the following deliverables:

1. all the source files required by your program (the command make should (still) create the executable main, which we will run to test your program)
2. a report (suggested length: about two pages) containing:
   (a) a general description of the data types your algorithm uses
   (b) and of how your algorithm works

(c) and of how you tested your program

(d) the description of the difficulties you have encountered

(e) description of the possible extensions you have added

(f) anything else you think we should know

3. optionally, test files which you think are particularly interesting

2 Abstract syntax

We consider the moral subset of ML given by the following abstract syntax:

$$
\begin{array}{lcl}
id & = & [a-zA-Z][a-zA-Z0-9]* \\
n & = & [0-9]+ \\
b & = & \textbf{true} \mid \textbf{false} \\
e & = &
\end{array}
$$

id	n	$e = e$	b
fun $id = e$	$e + e$	**ref** e	e **and** e
$e\ e$	$e - e$	**!**e	e **or** e
let $id = e$ **in** e	$e * e$	e **:=** e	e **xor** e
letrec $id = e$ **in** e	e / e	e**;** e	**not** e
()	e % e	(e)	**if** e **then** e **else** e
	$-e$	**type**(e)	
	$e \leq e$		
	$e \geq e$		
	$e < e$		
	$e > e$		

id denotes an identifier, n a natural number and b a boolean value. A program is an expression e. You can extract the semantics of such a program from the interpreter source code in the main.ml file. It is pretty much the standard call-by-value sematics you expect.

3 Typing

A type system is sound if a program that is typable returns a value, runs forever or terminates with a division by zero exception. You should implement the Damas-Milner-Tofte algorithm seen in the course to infer a sound type for the entire program. When you encounter an expression adnotated by "TYPE", print the type of the expression in the current typing environment.

If you encounter a type error you should print an error message and abord typing. Concentrate on printing pretty error messages **after** you finish everything else.

3.1 Polymorphism and references

You should pay particular attention to the interaction between the ML "let"-polymorphism and mutable cells. In particular, the following program should not be typable:

```
let f = ref (fun x = x) in
    ((f := fun x = x + 1); (!f) true)
```

You should research (e.g. try to see what happens in the ocaml toplevel or simply google things) how this problem is handled in practice and make sure your type inference program produces sound types.

4 Extensions

If you want, you may implement extensions. For example, some of you might want to add exceptions, subtyping, pairs, existential types, etc. However, if your extension is not conservative (some programs become untypable), please provide two versions of your project: the "standard" one and the extension-enabled version.

5 Examples

1. For the following program:

   ```
   let f = fun x = x in
      type(f)
   ```

   your program should print something along the lines of:

   ```
   forall a : a -> a
    a -> a
   ```

2. For the following program:

   ```
   type(
   let pair = fun a = fun b = fun f = f a b in
   /* let fst = fun p = p (fun x = fun y = x) in
   let snd = fun p = p (fun x = fun y = y) in */

   pair 10 12
   )
   ```

   the output should resemble:

```
(int -> int -> a) -> a
```

3. and the last example:

```
type(
let pair = fun a = fun b = fun f = f a b in
let fst = fun p = p (fun x = fun y = x) in
/*let snd = fun p = p (fun x = fun y = y) in */

fst (pair 10 12)
)
```

the output should resemble:

```
int
```

4. new example 1:

```
let pair = fun a = fun b = fun f = f a b in
let id = fun x = x in
pair type(id) type(id)
```

should output something like:

```
a1 -> a1
a2 -> a2
```

5. new example 2:

```
let f = ref (fun x = x) in
    ((f := fun x = x + 1); (!f) true)
```

should output something like:

```
untypable
```

You should also look at the input files that come with the scaffolding code.