Assignment 5
COMP 302 Programming Languages and Paradigms

Brigitte Pientka
MCGILL UNIVERSITY: School of Computer Science

Due Date: 28 Nov 2013

In this homework, we will implement a language called MiniML, in OCaml. This will allow us to explore key concepts such as free variables, substitution, evaluation, type checking and type inference. MiniML is similar to the language we discussed in class; it however also includes n-ary tuples (pairs written as $(e_1, \ldots,\ e_n)$), a more general let-expression, functions, function application, and recursion.

Hence our internal MiniML language now contains the following types and expressions:

types τ	:=	$\mathsf{int} \mid \mathsf{bool} \mid \tau_1 \text{ -> } \tau_2 \mid \tau_1 * \ldots * \tau_n$
expressions e	:=	$\ldots \mid \mathsf{fn}\ (x : \tau) \Rightarrow e \mid e_1\ e_2 \mid \mathsf{rec}\ (f : \tau) \Rightarrow e \mid (e_1,\ \ldots,\ e_2) \mid \mathsf{let}\ ds\ \mathsf{in}\ e\ \mathsf{end}$
declaration d	:=	$\mathsf{val}\ x = e \mid \mathsf{val}\ (x_1, \ldots, x_n) = e \mid \mathsf{name}\ x = e$
declarations ds	:=	$\cdot \mid d\ ds$

Our external language, i.e. the language the programmer uses and the parser accepts is slightly different. Most notably, we write for example

```
let fun fact (x : int) : int =
  if x = 0 then 1
  else x * fact(x - 1)
in
  fact 5
end;
```

which corresponds to

$$\mathsf{rec}\ (\mathsf{fact} : \mathsf{int} \text{ -> } \mathsf{int}) \Rightarrow \mathsf{fn}\ (x : \mathsf{int}) \Rightarrow \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ x * fact(x - 1)$$

To get started, download the template code from WebCT; you will find several files in the directory `mini-ml` which allow you to parse input files written in MinML, and print back programs in MinML, and the implementation of the interpreter has been described in class. The code can both be compiled to generate byte code or native code; this is the usual mode a MiniML programmer would use it to parse, type check and evaluate the programs she writes. You can also load the executable code into the interpreter, if you wish, to easily test individual functions. This might be useful for developers, since it allows for quicker code development.

Please read carefully through the `README` file before beginning this homework.

Q1 15 points OCaml also tests whether a given variable is used or not; this is a useful little tool. Implement a function `unusedVars: exp -> string list` which given an expression checks for each variable introduced by a binding construct (i.e. a function, let expression, recursion, etc.) whether the variable the construct introduces is in fact used in the body of the expression.

For example:

let val `x = 3` **in** `4` **end**	`x` is unused
let val `x = true` **in** **let val** `y = 4` **in** `x + 5` **end** **end**	`y` is unused
let val `x = 3` **in** **let val** `x = 4` **in** `x + x` **end** **end**	`x` (the first occurrence) is unused

Similarly, given the following program, `x` and `test` is unused:

let fun `test (x : int) : int = 3`

Implement the function `unusedVars: exp -> string list`. It traverses an expression and for every construct which introduces some bound variables (i.e. recursion, functions, let-expression) we will check whether these variables occur free in the body; if they do, then indeed these variables are used; if they do not occur free in the body, then the variables are unused. We already provided an implementation for computing the free variables.

Q2 (25 points) In this question, we will implement an interpreter based on *big-step evaluation*. Your task is to go into `eval.ml` and implement the missing cases of the function `eval` for big-step evaluation. In particular, add the cases for functions, function application, recursion, and handling generalized let-expressions.

$$e \Downarrow v \quad \text{expression } e \text{ evaluates to value } v$$

$$\frac{e \Downarrow v}{\text{let } \cdot \text{ in } e \text{ end} \Downarrow v}\ \text{b-no-dec} \qquad \frac{e1 \Downarrow v_1 \quad [v_1/x](\text{let decs in } e \text{ end}) \Downarrow v}{\text{let val } x_1 = e_1 \text{ decs in } e \text{ end} \Downarrow v}\ \text{B-LET-VAL}$$

$$\frac{[e_1/x](\text{let decs in } e \text{ end}) \Downarrow v}{\text{let name } x_1 = e_1 \text{ decs in } e \text{ end} \Downarrow v}\ \text{B-LETN}$$

$$\frac{e1 \Downarrow (v_1, \ldots, v_n) \quad [v_1/x_1, \ldots, v_n/x_n](\text{let decs in } e \text{ end}) \Downarrow v}{\text{let val } (x_1, \ldots, x_n) = e_1 \text{ decs in } e \text{ end} \Downarrow v}\ \text{B-LET-TUPLE}$$

$$\frac{\text{for all } i \;\; e_i \Downarrow v_i}{(e_1,\ \ldots,\ e_n) \Downarrow (v_1, \ldots, v_m)}\ \text{B-TUPLE} \qquad \frac{[\text{rec } (f : \tau) \Rightarrow e/f]e \Downarrow v}{\text{rec } (f : \tau) \Rightarrow e \Downarrow v}\ \text{B-REC}$$

$$\frac{}{\text{fn } (x : \tau) \Rightarrow e \Downarrow \text{fn } (x : \tau) \Rightarrow e}\ \text{B-FN} \qquad \frac{e_1 \Downarrow \text{fn } (x : \tau) \Rightarrow e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}\ \text{B-APP}$$

If you encounter a situation not covered by these rules, then you should **raise** `Stuck "message"` (with something more descriptive than `"message"`). See the other cases for examples of this.

The evaluator is **not** environment-based, so there is a direct translation from the above rules to code. A function `subst` has been provided.

As is often the case, not very much code is required.

$$\frac{\Gamma \vdash e \Rightarrow \text{bool} \qquad \Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_2 \Rightarrow \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \tau}\ \text{T-IF} \qquad \frac{\Gamma, x{:}\tau \vdash e \Rightarrow \tau'}{\Gamma \vdash (\text{fn } (x : \tau) \Rightarrow e) \Rightarrow (\tau \text{ -> } \tau')}\ \text{T-FN}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n \Rightarrow \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) \Rightarrow (\tau_1 * \cdots * \tau_n)}\ \text{T-TUPLE}$$

$$\frac{\Gamma, f{:}\tau \vdash e \Rightarrow \tau' \quad \tau = \tau'}{\Gamma \vdash (\text{rec } (f : \tau) \Rightarrow e) \Rightarrow \tau}\ \text{T-REC} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau \text{ -> } \tau' \qquad \Gamma \vdash e_2 \Rightarrow \tau}{\Gamma \vdash e_1\, e_2 \Rightarrow \tau'}\ \text{T-APP}$$

$$\frac{\Gamma \vdash \text{decs} \Rightarrow \Gamma' \qquad \Gamma, \Gamma' \vdash e \Rightarrow \tau}{\Gamma \vdash \text{let decs in } e \text{ end} \Rightarrow \tau}\ \text{T-LET} \qquad \frac{\Gamma \vdash e \Rightarrow \tau' \qquad \tau = \tau'}{\Gamma \vdash (e : \tau) \Rightarrow \tau}\ \text{T-ANNO}$$

Rules for declarations

$$\frac{\Gamma \vdash \text{dec}_1 \Rightarrow \Gamma_1 \qquad \Gamma, \Gamma_1 \vdash \text{decs} \Rightarrow \Gamma_2}{\Gamma \vdash \text{dec}_1 \text{ decs} \Rightarrow \Gamma_1, \Gamma_2}\ \text{T-DECS}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\text{val } x = e) \Rightarrow (x : \tau)}\ \text{T-BY-VAL} \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\text{name } x = e) \Rightarrow (x : \tau)}\ \text{T-BY-NAME}$$

$$\frac{\Gamma \vdash e \Rightarrow (\tau_1 * \cdots * \tau_n)}{\Gamma \vdash (\text{val } (x_1, \ldots, x_n) = e) \Rightarrow (x_1 : \tau_1), \ldots, (x_n : \tau_n)}\ \text{T-BY-VAL-TUPLE}$$

Figure 1: Type inference rules

Q3 (30 points) In this question, we are inferring the type of an expression. We use the type annotations on recursion and functions, to guarantee that we can always uniquely infer a unique type of an expression. It simplifies the type inference problem. For now, you can ignore type variables.

To describe the type inference we will use the following judgement:

$$\Gamma \vdash e \Rightarrow T \qquad \text{Infer type } T \text{ for expression } e \text{ in context } \Gamma$$

The inference rules are given in Figure 1. Your implementation of the function `infer: context -> Minml.exp -> Type.tp` should follow exactly the algorithm described by the typing rules from Figure 1. It takes as input a context describing the typing assumptions and an expression. It returns the type of the expression.

Go to the directory `mini-ml`, and implement the function `infer` in the file `typecheck.ml`.

```
type tp =  (* MiniML Types *)
    | Int  (* T ::= Int *)
    | Bool  (* | Bool *)
    | Arrow of tp * tp  (* | T1 => T2 *)
    | Product of tp * tp  (* | T1 * T2 *)
    | TVar of (tp option) ref  (* IGNORED FOR NOW *)
```

See the directory `examples` for some examples. For how to test your functions, please see the `README` file.

Q4 (25 points) Lastly, we want to enable full type inference; as a consequence, we can omit the type annotations on functions and recursion. Key to type inference is unification. Your task is to implement the function `unify:Type.tp * Type.tp -> unit` in the file `unify.ml`, which checks whether two types are unifiable, i.e. if we can make them syntactically equal.

You should follow the description of unification in the class notes. Type variables are modeled via references. An uninstantiated type variable is modeled as a pointer to a cell with content `None`. In other words to create a new type variable we can simply use a function

```
let freshVar () = TVar (ref None)
```

Once we know what a type variable should be instantiated with we simply assign it the correct type. For example, if we have a type variable `TVar x`, then `x` has type `(tp option) ref` and we can replace every occurrence of `x` by the type `Int` using assignment `x := Some (Int)`.

This will destructively update the type variable `x` and directly propagate the instantiation for it. No extra implementation of a substitution function is necessary to propagate instantiations.

Your task is to implement the function `unify:Type.tp * Type.tp -> unit` which tests whether two types are unifiable. If two types are unifiable, they will be denoting the same type after unification succeeds. If unification fails, raise an exception. Follow the algorithm described in the notes to unify two types, and fill in the implementation for `unify` in file `unify.ml`.

We have provided a file `unify_test.ml` which shows you how to test the `unify` and verify it is working correctly.

Q5 (10 points) **OPTIONAL - EXTRA CREDIT** Modify your function `infer` so that it supports type inference. This is quite easy: if the type annotation in recursion and functions is `None` and we don't know the type of a variable `x` we want to add to the context, we simply generate a fresh type variable `tv` and add to the context the fact that `x` has type `tv`; instead of checking for equality, you call unification.

Q6 (5 points) It is time to think about studying for the final! Asking questions about the material you've learned is an important tool to check whether you have understood the material and reflect on the material you've seen. It is also a great way to study for exams. Please submit one question about the material together with a (hopefully) correct answer which in your opinion could be used on an exam. Your question can be theoretical in nature (concerning inductive proofs, environment diagrams, substitution definitions, free variable definition, inferring the type for an expression, etc.) or be concerned with writing a program (using higher-order functions, references for modelling objects, exceptions, continuations for backtracking and for writing tail-recursive functions, lazy programming, etc.). Write up your question in an ascii-file called `question.txt` and submit it via handin. We will then collect your questions and make them accessible to your fellow students.

Q7 (0 points) Please take a moment to fill out the course evaluations!