Homework 4

Due: June 28, 2015, 5.00 pm. Submit PDF + code files on Moodle.

1 Theoretical Questions (25 points)

Objectives

The purpose of this HW is to test your understanding of how to use typing rules to perform polymorhic type derivations in a functional programming language (here with OCaml syntax). Another purpose of HWs is to provide you with experience answering non-programming written questions of the kind you may experience on the midterms and final.

What to Turn In

Your answers to the following questions are to be submitted electronically via the Moodle. You should type up your answers to each of the problems below and submit them in a file called `hw4.pdf`. You can use LaTeX or any other wordprocessor of your choice to type up the solutions but we can only grade PDF files. (You can even write your answers by hand legibly. You should then make a reasonable effort to make a readable scan of it, and submit those documents as PDF file; please don't submit raw image files.)

We shall only grade PDF submissions. We will ask you to resubmit solutions in any other file format.

Problems

Problem 1: (25 points)

Give a complete type derivation for the following typing judgment.

```
let rec f = fun x -> fun n -> if n <= 0 then [] else x::(f x (n - 1))
       in (f 3 2, f "a" 4) : int list * string list
```

As a suggestion for formatting, you may want to name subtrees of the proof and write them out separately. Note, we are asking for a type derivation not the intermediate states of a type inferencing algorithm.

2 Machine Problems (42 points)

Objectives

The purpose of this MP is to test the student's ability to

- constructing algebraic data types
- deconstructing algebraic data types
- continuation passing style transformations

Another purpose of MPs in general is to provide a framework to study for the exam. Several of the questions on the exam will appear similar to the MP problems.

What to Turn In

You should put code answering each of the problems below in a file called `mp4.ml`. A good way to start is to copy `mp4-skeleton.ml` to `mp4.ml` and edit the copy. Please read the Guide for Doing MPs.

Background

Throughout this MP we will be working with a (very) simple functional language. It is the seed of the language that we will use in MPs throughout the rest of this semester. In this MP, instead of writing our programs in text files and parsing them, we will represent the structure of our programs via terms made from a set of four algebraic data types.

In this MP, you will primarily be working the data type `exp`, which we will describe here. It is the main type representing expressions in our simple programming language. The type `exp` makes use of three other data types. The type `const` describes the type of values in our language. We allow for integers, booleans, and the empty list. This set will be expanded in later assignments.

```
type const = Int of int | Bool of bool | Nil
```

The types `monop` and `binop` represent the names of built-in operations of one or two arguments. The binary operations supported here are addition, subtraction and multiplication on the integers, generic equality and ordering testing, and consing of elements onto a list. Again, these types will grow in later assignments. The operators of one argument are for taking the head and the tail of a list and printing integer values. The operators for taking the head and tail of a list must be included among the primitive operations because we have no pattern-matching in this simple language.

```
type binop = Add | Sub | Mul | Eq | Less | Cons
```

```
type monop = Head | Tail | Print
```

The next data type `exp` gives the ways we have of making expressions in our language: variables and constants, if-then-else expressions, application of one expression to another, expression using built in operations of one or two arguments, functions expressions, local `let`-bindings, and recursive local `let rec`-bindings.

```
type exp =
   | VarExp of string                    (* variables *)
   | ConExp of const                     (* constants *)
   | IfExp of exp * exp * exp            (* if exp1 then exp2 else exp3 *)
   | AppExp of exp * exp                 (* exp1 exp2 *)
   | BinExp of binop * exp * exp         (* exp1 % exp2
                                            where % is a builtin binary operator *)
   | MonExp of monop * exp               (* % exp1
                                            where % is a builtin monadic operator *
   | FunExp of string * exp              (* fun x -> exp *)
   | LetExp of string * exp * exp        (* let f x = exp1 in exp2 *)
   | RecExp of string * string * exp * exp    (* let rec x = exp1 in exp2 *)
   | OAppExp of exp * exp                (* Extra credit *)
```

An example of the use of this data type would be to represent the function calculating the length of a list:

```
let rec length l = if l = [] then 0 else 1 + (length (tl l)) in length
```

becomes

```
RecExp("length", "l",
      IfExp(BinExp(Eq,VarExp "l", ConExp Nil),
           ConExp(Int 0),
           BinExp(Add, ConExp(Int 1),
                      AppExp(VarExp "length", MonExp(Tail, VarExp "l")))),
      VarExp "length")
```

To facilitate in debugging your code, in the module `Mp4common` we have supplied you with a function `string_of_exp : Mp4common.exp -> string` that will generate the concrete syntax that corresponds to a given `exp` term. For example, if you apply `string_of_exp` to the `exp` term immediately above, you get a string containing the code displayed immediately before that. We also provide a function `eval : exp -> string` that will "execute" your code, generating a string that is what the top-level loop would print as a value if you were to execute the corresponding code in OCaml. For example, calling the `exp` version of `length`:

```
# eval (RecExp("length","l",...,AppExp(VarExp "length",ConExp(Nil))));;
- : string = "0"
```

To use `eval`, build it and then import the needed modules:

```
\% make top
\% ./mp4-top
open Mp4common;;
open Student;;
open Mp4eval;;
open Rubric;;
# eval (RecExp("length","l",...,AppExp(VarExp "length",ConExp(Nil))));;
- : string = "0"
```

Problems

Note: In the problems below, you do not have to begin your definitions in a manner identical to the sample code, which is present solely for guiding you better. However, you have to use the indicated name for your functions and values, and they will have to conform to any type information supplied, and have to yield the same results as any sample executions given, as well as satisfying the specification given in English.

The problems below have sample executions that suggest how to write answers. Students have to use the same function name, but the name of the parameters that follow the function name need not be duplicated. That is, the students are free to choose different names for the arguments to the functions from the ones given in the example execution. You are not required to start your code with `let rec`. You may use any library functions you wish.

Problem 1: (4 points)

Write a function `import_list: int list -> exp`, that takes an integer list and converts it into an expression in our language that is equivalent.

```
# let rec import_list lst = ... ;;
val import_list : int list -> Mp4common.exp = <fun>
# import_list [1;2];;
- : exp = BinExp(Cons,ConExp (Int 1),BinExp(Cons,ConExp (Int 2),ConExp Nil))
```

Problem 2: (4 points)

Write a term in our language that implements the `elem` function, which decides if the first argument is an element of the given list, using the following OCaml code:

```
let rec elem e = fun xs -> if xs = [] then false
                        else if hd xs = e then true
                        else elem e (tl xs)
 in elem
```

For this code to actually compile in OCaml, `open List;;` would first have to be executed.

```
# let elem = ... ;;
val elem : exp = ...
# string_of_exp elem;;
- : string =
"let rec elem e = fun xs -> if xs = [] then false else if hd xs = e
then true else elem e (tl xs) in elem"
```

You can test out your implementation by evaluating it on various input as follows:

```
# #load "mp4eval.cmo";;
# open Mp4eval;;
# eval (AppExp(AppExp(elem,ConExp (Int 1)),import_list [1;2;3]));;
- : string = "true"
# eval (AppExp(AppExp(elem,ConExp (Int 4)),import_list [1;2;3]));;
- : string = "false"
```

Problem 3: (12 points)

Write a function `num_of_consts : exp -> int` that counts the number of occurrences of the constructor `ConExp` in an `exp`.

```
# let rec num_of_consts exp = ...
val num_of_consts : exp -> int = <fun>
# num_of_consts (IfExp(BinExp(Eq,VarExp "l", ConExp Nil), ConExp(Int 0),
                  VarExp "x"));;
- : int = 2
```

Problem 4: (10 points)

A free variable in an expression is a variable that isn't bound in that expression. Free variables are the variables that had to be given a value previously for the expression to be able to be evaluated. As an example, in (`let x = y in fun s -> a x s`) the variables `a` and `y` are free but `x` and `s` are not.

Write a function `freeVars : exp -> string list` that calculates the names of the free variables of an expression. As in MP2, represent sets via lists. The grader will cope with answers that have duplicate entries or the result list in a different order than our reference solution. You may notice that the case for `OAppExp` (which we will write infixed as $(e_1 \$ e_2)$) is missing; that will be covered in the extra credit.

To assist you in writing this function, we have broken the problem down into groups of similar cases. We also give the precise mathematical definition (in cases) for a function φ calculating the free variables of an expression e.

a. (1 point) We can define a function $\varphi(e)$ that calculates the free variables of an expression, where the expression is a variable v, or a constant c by

$$\varphi(v) = v$$
$$\varphi(c) = \emptyset$$

The function `freeVars` should behave in a similar manner, returning no names for a constant, and the singleton name of a variable. Write the appropriate clause for `freeVars` to return the free variables of expressions that are constants or variables.

```
# let rec freeVars = ... ;;
val freeVars : exp -> string list = <fun>
# freeVars (VarExp "x");;
- : string list = ["x"]
```

b. (4 points) The set of free variables of an expression that is top-most an if-then-else, the use of a unary or binary operator, or the application of one expression to another is just the union of the free variables of all the immediate subexpressions.

$$\begin{aligned} \varphi(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) &= \varphi(e_1) \cup \varphi(e_2) \cup \varphi(e_3) \\ \varphi(\oplus e) &= \varphi(e) && \text{For unary operator } \oplus \\ \varphi(e_1 \oplus e_2) &= \varphi(e_1) \cup \varphi(e_2) && \text{For binary operator } \oplus \\ \varphi(e_1\ e_2) &= \varphi(e_1) \cup \varphi(e_2) \end{aligned}$$

Write the clauses for `freeVars` for expressions that are top-most an if-then-else, the use of a unary or binary operator, or the application of one expression to another.

```
# freeVars (IfExp(ConExp(Bool true), VarExp "x", VarExp "y"));;
- : string list = ["x"; "y"]
```

c. (1.5 points) The free variables of a function expression are all the free variables in the body of the expression except the variable that is the formal parameter. Any occurrence of that variable in the body of the function is bound by the formal parameter, and not free.

$$\varphi(\texttt{fun}\ x\ \texttt{->}\ e) = \varphi(e) - \{x\}$$

Add clauses to `freeVars` to compute the free variables of a function expression.

```
# freeVars (FunExp("x", VarExp "x"));;
- : string list = []
```

d. (1.5 points) The free variables of a `let`-expression are also restricted by the binding the `let` introduces. In `let` x `=` e_1 `in` e_2 the x bind any occurrence of x in e_2, as in the body of a function, but does not change which variables free in e_1 are free in the whole expression.

$$\varphi(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) = \varphi(e_1) \cup (\varphi(e_2) - \{x\})$$

Add the clause to `freeVars` to compute the free variables of `let`-expressions.

```
# freeVars (LetExp("x", VarExp "y", VarExp "x"));;
- : string list = ["y"]
```

e. (2 points) The most complicated case for computing the free variables of an expression is that of a `let rec`-expression. In `let rec`-expressions, there are two bindings taking place, and they have two different scopes. In `let rec` f x `=` e_1 `in` e_2`)`, the f binds all the occurrences of f in both e_1 and e_2, but the x only binds occurrences of x in e_1; if x is a free variable of e_2 it will be a free variable of `let rec` f x `=` e_1 `in` e_2`)`.

$$\varphi(\texttt{let rec}\ f\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\texttt{)} = (\varphi(e_1) - \{f, x\}) \cup (\varphi(e_2) - \{f\})$$

Write the clause for `freeVars` for `let rec`-expressions.

```
# freeVars (RecExp("f","x",AppExp(VarExp "f",VarExp "x"),
                (AppExp(VarExp "f",VarExp "y"))));;
- : string list = ["y"]
```

Problem 5: (8 points)

In MP3 you converted some expressions to use Continuation-Passing Style (CPS). In this section you will build a function `cps : exp -> exp -> exp` to automatically transform expressions in our language into CPS.

MAthematically we represent this transformation by the function $[[e]]_\kappa$, which calculates the CPS form of an expression e when passed the continuation κ. κ does not represent a programming language variable, but rather a complex expression describing the current continuation for e.

The defining equations of this function are given below. In these rules f, k, x, v and v_i represent variables in our programming language, c is a constant, e or e_i are expression. The variables f and x will represent variables that were already present in the expression to be transformed. The variables v and v_i are used to represent newly introduced variables used to pass a value from the previous computation forward into the current continuation. The variable k is used to represent a variable (such as a formal parameter to a function) to be instantiated by an as yet unknown continuation.

By v being fresh for an expression e, we mean that v needs to be some variable that is NOT free in e. In `Mp4common`, we have supplied a function `freshFor : string list -> string` that, when given a list of names, will generate a name that is not in the list. When implementing `cps`, the names you use for these "fresh" variables do not have be the same as the ones we use, but they do have to satisfy the required freshness constraint.

a. (1 point) The CPS transformation of a variable or constant expression just applies to the continuation to the variable or constant, since during execution, when this point in the code is reached, both variables and constants are already fully evaluated (except for being looked up).

$$[[v]]_\kappa = \kappa\ v$$
$$[[c]]_\kappa = \kappa\ c$$

The code for the function `cps` should behave is a similar manner, creating the application of the continuation to the variable or constant. Add code to `cps` to implement the CPS-transformation of an expression that is a variable or constant.

```
# string_of_exp (cps (VarExp "x") (VarExp "k"));; - : string = "k x"
```

b. (1 point) Each CPS transformation should make explicit the order of evaluation of each subexpression. For if-then-else expressions, the first thing to be done is to evaluate the boolean guard. The resulting boolean value needs to be passed to an if-then-else that will choose a branch. When the boolean value is true, we need to evaluate the transformed then-branch, which will pass its value to the final continuation from the if-then-else expression. Similarly, when the boolean value is false we need to evaluate the transformed else-branch, which will pass its value to the final continuation from the if-then-else expression. To accomplish this, we recursively CPS-transform e_1 with the continuation with a formal parameter v, that is fresh for e_2, e_3 and κ, where, based on the value of v, the continuation chooses either the CPS-transform of e_2 with the original continuation κ, or the CPS-transform of e_3, again with the original continuation κ.

$$[[\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]]_\kappa = [[e_1]]_{\texttt{fun}\ v\ \texttt{->}\ \texttt{if}\ v\ \texttt{then}\ [[e_2]]_\kappa\ \texttt{else}\ [[e_3]]_\kappa}$$

Where v is fresh for e_2, e_3, and κ

Add a clause to `cps` for the case for if-then-else operators.

```
# string_of_exp (cps (IfExp (VarExp "b", ConExp (Int 2), ConExp (Int
5))) (VarExp "k"));; - : string = "(fun a -> if a then k 2 else k 5)
b"
```

c. (1 point) The CPS transformation for application mirrors its evaluation order. It first evaluates the function, e_1, to a closure then evaluates e_2 to a value which that closure is applied to. We create a new continuation that takes the result of e_1 binds it to v_1 then evaluates e_2 and binds it to v_2. Finally, v_1 is applied to v_2 and, since the CPS transformation makes all functions take a continuation, to the current continuation κ. Implement this rule.

$$[[e_1\ e_2]]_\kappa = [[e_1]]_{\texttt{fun}\ v_1\ \texttt{->}\ [[e_2]]_{\texttt{fun}\ v_2\ \texttt{->}\ v_1\ v_2\ \kappa}}$$

$$\text{Where}\ \begin{array}{l} v_1 \text{ is fresh for } e_2 \text{ and } \kappa \\ v_2 \text{ is fresh for } v_1 \text{ and } \kappa \end{array}$$

```
# string_of_exp (cps (AppExp (VarExp "f", VarExp "x")) (VarExp
"k"));; - : string = "(fun a -> (fun b -> a b k) x) f"
```

d. (1 point) The CPS transformation for a binary operator mirrors its evaluation order. It first evaluates its first argument then its second before evaluating the binary operator applied to those two values. We create a new continuation that takes the result of the first argument, e_1, binds it to v_1 then evaluates the second argument, e_2, and binds that result to v_2. As a last step it applies the current continuation to the result of $v_1 \oplus v_2$. Implement the following rule.

$$[[e_1 \oplus e_2]]_\kappa = [[e_1]]_{\texttt{fun}\ v_1\ \texttt{->}\ [[e_2]]_{\texttt{fun}\ v_2\ \texttt{->}\ \kappa\ (v_1 \oplus v_2)}}$$

$$\text{Where}\ \begin{array}{l} v_1 \text{ is fresh for } e_1,\ e_2, \text{ and } \kappa \\ v_2 \text{ is fresh for } e_1,\ e_2,\ \kappa, \text{ and } v_1 \end{array}$$

```
# string_of_exp (cps (BinExp (Add, ConExp(Int 5), ConExp(Int 1)))
                    (VarExp "k"));;
- : string = "(fun a -> (fun b -> k (a + b)) 1) 5"
```

e. (1 point) The CPS transformation for a unary operator mirrors its evaluation order. It first evaluates the argument of the operator and then applies the continuation to the result of applying that operator to the value. Thus we create a continuation that takes the result of evaluating the argument, e, and binds it to v then applies the continuation to the result of $\oplus v$. Implement the following rule.

$$[[\oplus e]]_\kappa = [[e]]_{\texttt{fun}\ v\ \texttt{->}\ \kappa\ (\oplus v)} \qquad \text{Where } v \text{ is fresh for } \kappa$$

```
# string_of_exp (cps (MonExp (Head, ConExp Nil)) (VarExp "k"));; - :
string = "(fun a -> k (hd a)) []"
```

f. (1 point) A function expression by itself does not get evaluated (well, it gets turned into a closure), so it needs to handed to the continuation directly, except that when it eventually gets applied, it will need to additionally take a continuation as another argument, and its body will need to have been transformed. Therefore, we need to choose a variable k that is fresh for the body of the function, e, to be the formal parameter for passing a continuation into the function. Then, we need to transform the body with k as its continuation, and put it inside a function with the same original formal parameter together with k. The original continuation κ is then applied to the result.

$$[[\texttt{fun}\ x\ \texttt{->}\ e]]_\kappa = \kappa\,(\texttt{fun}\ x\ \texttt{->}\ \texttt{fun}\ k\ \texttt{->}\ [[e]]_k) \qquad \text{Where } k \text{ is fresh for } e$$

Write the clause for the case for functions.

```
# string_of_exp (cps (FunExp ("x", VarExp "x"))
                   (VarExp "k"));;
- : string = "k (fun x -> fun a -> a x)"
```

g. (1 point) A `let` expression first evaluates the expression being bound, e_1, binds it to the name x and then evaluates e_2 in the context of that new binding. You may notice that this is roughly what a function does during evaluation. For example, `let` x `=` e_1 `in` e_2 could have been written as (`fun` x `->` e_2)e_1. To transform a `let` expression to CPS we construct a continuation that takes the result of evaluating e_1 binds it to x then evaluates e_2 with this new binding, passing along the current continuation. Implement the following rule.

$$[[\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2]]_\kappa = [[e_1]]_{\texttt{fun } x \texttt{ -> } [[e_2]]_\kappa}$$

```
# string_of_exp (cps (LetExp ("x", ConExp(Int 2), VarExp "x"))
                   (VarExp "k"));;
- : string = "(fun x -> k x) 2"
```

h. (1 point) A `let rec` expression creates recursive definition for f and then evaluates the body, e_2, with this definition in scope. Since we require `let rec` expressions to be functions we do the CPS transform for the function binding as well as passing the current continuation to the body. Implement the following rule.

$$[[\texttt{let rec } f\ x \texttt{ = } e_1 \texttt{ in } e_2]]_\kappa = \texttt{let rec } f\ x \texttt{ = fun } v \texttt{ -> } [[e_1]]_v \texttt{ in } [[e_2]]_\kappa$$

Where v is fresh for f, x, and e_1

```
# string_of_exp (cps (RecExp ("f","x",VarExp "x",ConExp (Int 4))) (VarExp "k"));;
- : string = "let rec f x = fun a -> a x in k 4"
```

Problem 6: (4 points)

The `OAppExp` constructor for our language is application that evaluates its second argument first and then its first before applying the first to the second. It introduces no new bindings. Add the (e_1\$$e_2$) case for both `freeVars` and `cps`.

```
# string_of_exp (cps (OAppExp (FunExp ("x", VarExp "x"), ConExp (Int
2))) (VarExp "k"));; - : string = "(fun a -> (fun b -> b a k) (fun x
-> fun a -> a x)) 2"
```

3 Guide for Doing MPs

a. Setup a directory for CS 421 and a subdirectory for HWs/MPs.

b. Create a subsubdirectory corresponding to this HW within them.

c. Retrieve the directory for this MP posted on Moodle and all its contents. Get into that directory. (If we revise an assignment, you will need to repeat this to obtain the revision.)

d. To make sure you have all the necessary pieces, start by executing make. This will create the grader executable. Run the executable (`>$ ./grader`). Examine the failing test cases for places where errors were produced by your code. At this point, everything should compile, but the score will be 0.

e. Read and understand the problem for the handout on which you wish to begin working. (Usually, working from top to bottom makes most sense.) There is a tests file in this directory. This is an important file containing the an incomplete set of test cases; you'll want to add more cases to test your code more thoroughly. Reread the problem from the handout, examining any sample output given. Open the tests file in the mpX directory. Find the test cases given for that problem. Add your own test cases by following the same pattern as of the existing test cases. Try to get a good coverage of your function's behaviour. You should even try to have enough cases to guarantee that you will catch any errors. (This is not always possible, but a desirable goal.) And yes, test cases should be written even before starting the implementation of your function. This is a good software development practice.

f. If necessary, reread the statement of the problem once more. Place your code for the solution in mpX.ml (or mpX.mll or mpX.mly as specified by the assignment instructions) replacing the stub found there for it. Implement your function. Try to do this in a step-wise fashion. When you think you have a solution (or enough of a part of one to compile and be worth testing), save you work and execute make and the ./grader again. Examine the passing and failing test cases again. Each failure is an instance where your code failed to give the right output for the given input, and you will need to examine your code to figure out why. When you are finished making a round of corrections, run make, followed by ./grader again. Continue until you find no more errors. Consider submitting your partial result so that you will at least get credit for what you have accomplished so far, in case something happens to interfere with your completing the rest of the assignment.

g. When your code no longer generates any errors for the problem on which you were working, return to steps 3) and 4) to proceed with the next problem you wish to solve, until there are no more problems to be solved.

h. When you have finished all problems (or given up and left the problem with the stub version originally provided), you will need to submit your file along with your PDF submission on Moodle.