Handout 3
CS242: Autumn 2004
13 October

Homework 3

Due 20 October

Reading

1. Read chapters 5(Algol and ML) and 6 (Types) of the text.

Problems

1. .... Algol 60 Procedure Types

In Algol 60, the type of each formal parameter of a procedure must be given. However, *proc* is considered a type (the type of procedures). This is much simpler than the type of a function in most languages, since function types usually give the type of the function arguments and the type of the return result. In fact, this is a loophole in Algol 60 compile-time type checking: since calls to procedure parameters are not fully type checked, Algol 60 programs may produce run-time type errors.

Write a procedure declaration for `Q` which causes the following program fragment to produce a run-time type error.

```
proc P (proc Q)
   begin Q(true) end;
P(Q);
```

where `true` is a boolean value. Explain why the procedure is statically type correct in Algol 60, but produces a run-time type error. (You may assume that adding a boolean to an integer is a run-time type error.)

2. .... ML Reduce for Trees

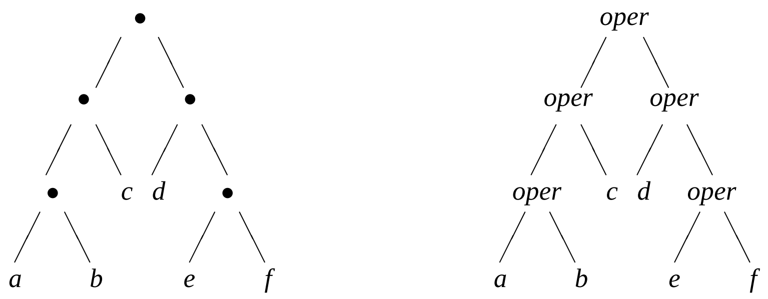
The binary tree datatype

$$\begin{aligned}\texttt{datatype}\ '\texttt{a tree} \ &= \ \texttt{LEAF of}\ '\texttt{a}\ | \\ &\ \ \texttt{NODE of}\ '\texttt{a tree} * '\texttt{a tree};\end{aligned}$$

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node). Write a function

$$\texttt{reduce} : ('\texttt{a} * '\texttt{a} \to '\texttt{a}) \to '\texttt{a tree} \to '\texttt{a}$$

that combines all the values of the leaves using the binary operation passed as a parameter. In more detail, if `oper` : $'\texttt{a} * '\texttt{a} \to '\texttt{a}$ and `t` is the nonempty tree on the left in this picture,



then `reduce oper t` should be the result obtained by evaluating the tree on the right. For example, if `f` is the function

$$\texttt{fun f(x : int, y : int) = x + y;}$$

then $\texttt{reduce f (NODE(NODE(LEAF 1, LEAF 2), LEAF 3))} = (1+2)+3 = 6$. Explain your definition of `reduce` in one or two sentences.

3. . . . . ML implementation of lambda reduction

This exercise asks you to implement β-reduction in ML and test your reduction function. The next few paragraphs explain the datatype you should use and give you some example terms. For the parts of the question that ask you to fill in missing parts of code we supply, you may turn in only the lines that contain the missing parts. For the part of the problem that asks for a complete function, please turn in the complete code for your function. This function can be written in less than 15 lines of code. We wont be too picky about coding aesthetics – the point of this problem is simply to ask you to try to "think" in ML a bit.

Lambda terms with addition have the BNF

$$t \quad ::= v \,|\, c \,|\, t+t \,|\, t\,t \,|\, \lambda v.t$$

In words, a lambda terms is either a variable, a constant (symbol with a fixed meaning), a sum of two terms, an application of one term to another, or a lambda abstraction. It is assumed that there are infinitely many variables, $v_1, v_2, v_3, \ldots$ so we can write a term with as many variables as we need, and there are enough "unused" variables that we can always α-convert when we need to. Since the only operation built into this version of lambda calculus is addition, we might as well assume that the constants are numbers $0, 1, 2, \ldots$. Some examples of terms produced by this grammar are $\lambda y.\, y+x$ and $\lambda y.\, \lambda x.\, y(yx)$

We can write a very similar ML datatype for lambda terms with addition, representing the variables $v_1, v_2, v_3, \ldots$ in the form $\texttt{Var}(1), \texttt{Var}(2), \ldots$ and the numbers $0, 1, 2, \ldots$ in the form $\texttt{Const}(0), \texttt{Const}(1), \texttt{Const}(2), \ldots$, as follows

```
datatype term =  Var of int| Const of int | Plus of term * term
              |  App of term * term | Lambda of int* term
```

Here are some example terms that can be used to make larger terms more readable.

```
val x = Var(1);
val y = Var(2);
val one = Const(1);
val two = Const(2);
```

Using the convenient declarations `val x_ = 1; val y_ = 2;` that let us write the numbers 1 and 2 using symbols that remind us of variables `x` and `y`, we can write terms $\lambda y.\, y+x$ and $\lambda y.\, \lambda x.\, y(yx)$, along with the application of the second term to the first, as

```
val twice = Lambda(y_, Lambda(x_, App(y, App(y,x))));
val addx = Lambda(y_, Plus(y,x));
val test = App(twice,addx);
```

You might notice that `test` is the lambda term used in lecture to illustrate the need for renaming bound variables when performing β-reduction.

(a) Write a function that substitutes a term for a variable, without renaming bound variables. Your function should have the form below, with the missing parts indicated by `(*-- missing part --*)` filled in. The function `subst{t,x,s)` substitutes `t` for all free occurrences of `x` in `s` and is defined by cases on the form of `s`:

```
fun subst(t,x,Var(n)) = if x=n then t else Var(n)
|   subst(t,x,Const(n)) = Const(n)
|   subst(t,x,Lambda(n,t1)) = if x=n then (*-- missing part --*)
                                     else (*-- missing part --*)
|   subst(t,x,App(t1,t2)) =  (*-- missing part --*)
|   subst(t,x,Plus(t1,t2)) = (*-- missing part --*)
```

(b) Write a function `ev` that evaluates a terms using `subst`. This function should not rename bound variables. Use the following form which breaks the problem into separate cases. In each case, evaluate all the parts that you can, but just return an expression like `ev (App(Const(3),Const(4)))` as is since there is no way to reduce this further. However, when an expression adds two numbers, you should return their sum.

```
fun ev(Var(n)) = Var(n)
|   ev(Const(n)) = Const(n)
|   ev(Lambda(n,t1)) = Lambda(n,ev(t1))
|   ev(App(t1,t2)) =
        (case ev(t1) of
              Var(n) => App(Var(n),ev(t2))      |
            Const(n) => App(Const(n),ev(t2))    |
         Lambda(n,t) => (*-- missing part --*)  |
           App(t,t1) => App(App(t,t1),ev(t2))   |
          Plus(t,t1) => (*-- missing part --*)
|   ev(Plus(t1,t2)) =
         case ev(t1) of
              Var(n) => Plus(Var(n),ev(t2))      |
            Const(n) => (case ev(t2) of
                            Var(m) => Plus(Const(n),Var(m))      |
                          Const(m) => (*-- missing part --*)     |
                       Lambda(m,t) =>  Plus(Const(n),Lambda(m,t))|
                         App(t3,t4) => (*-- missing part --*)    |
                        Plus(t3,t4) => Plus(Const(n),Plus(t3,t4)) )|
         Lambda(n,t) => Plus(Lambda(n,t),ev(t2)) |
           App(t,t1) => Plus(App(t,t1),ev(t2))   |
          Plus(t,t1) => (*-- missing part --*)   ;
```

Note that the case given for `ev(Lambda(n,t1))` evaluates the body of a lambda expression.

(c) Write a function `alpha` that renames bound variables so that no free variable in `alpha(term)` is the same as any bound variable. One way to do this is define
`alpha(term) = alph( ... ,term, ...)` where `alph` has additional arguments besides the term itself. For example, you could try something like
`alpha(term) = alph(free_variables(term),term)` where `free_variables` is a function that returns a list of free variables in a term. The idea here might be to use the list of free variables so that you can rename bound variables to names that are not in the list. There might also be some way to use the fact that variables are numbered to simplify this. You can be creative, but use your creativity to write a short, easy to read function, not some complicated mess. There is no need for assignment or reference cells, so do not use them.

(d) Define `eval(term) = ev(alpha(term))` and compare `ev(test)` with `eval(test)`.

(e) (*Extra Credit*) Consider replacing `ev(Lambda(n,t1)) = Lambda(n,ev(t1))`
with `ev(Lambda(n,t1)) = Lambda(n,t1)`. How is the resulting new definition of `eval` better? Worse?

4. .... ML Types

Explain the ML type for each of the following declarations:

(a) `fun a(x) = 3*x;`

(b) `fun b(x,y) = x/2.0 * y/3.0;`

(c) `fun c(f,g) = fn x => f(g(x));`

(d) `fun d(b,f,x,y) = if b(y) then f(x) else y;`

Since you can simply type these expressions into an ML interpreter to determine the type, be sure to write a short *explanation* to show that you understand why the function has the type you give.

5. . . . . Polymorphic Sorting

This function performing insertion sort on a list takes as arguments a comparison function `less` and a list `l` of elements to be sorted. The code compiles and runs correctly.

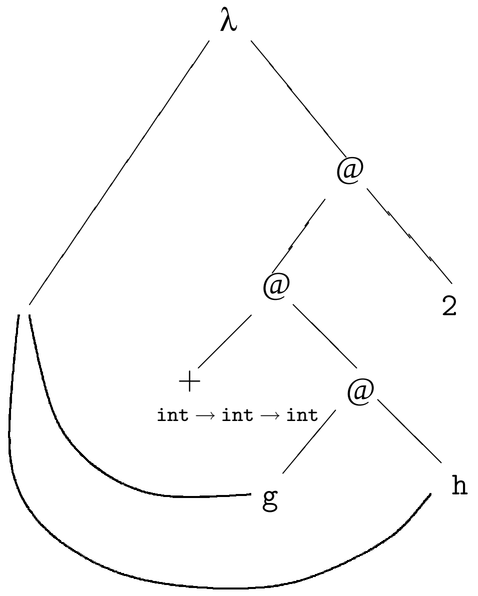
```
fun sort(less,nil) = nil |
    sort(less, a::l) =
       let
          fun insert(a, nil)  = a::nil |
              insert(a, b::l) = if less(a,b) then a::(b::l)
                                              else b::insert(a,l)
       in
          insert(a,sort(less,l))
       end;
```

What is the type of this `sort` function? Explain briefly, including the type of the subsidiary function `insert.` You do not have to run the ML algorithm on this code; just explain why an ordinary ML programmer would expect the code to have this type.

6. . . . . Parse Graph

(a) Use the parse graph below to calculate the ML type for the function

```
fun f(g,h) = g(h) + 2;
```



(b) Assuming that `f` has the type you found in part (a), draw the parse graph and perform ML type inference on `f(fn x=>x, 3)`.

7. . . . . Type Inference and Bugs

What is the type of the following ML function?

```
fun append(nil, l) = l
|   append(x::l, m) = append(l,m);
```

Write one or two sentences to explain succinctly and informally why `append` has the type you give. This function is intended to append one list onto another. However, it has a bug. How might knowing the type of this function help the programmer to find the bug?