

## Assignment 8

### Problem 1: Implementation of delay evaluation (call-by-name)

The SML interpreters you have written before implement call-by-value parameter passing, in which variables are evaluated as soon as they are defined. This results in eager behavior, as you’ve seen in SML. We will continue our exploration of interpreters by writing an interpreter for the same language as before (with the same representations), but with a different evaluation strategy (call-by-name). To implement call-by-name, in which variables are not evaluated until they are needed (which results in “lazy” behavior) we need to modify our rules:

1. For dynamic scope, we have:

$$(9a) \frac{\text{env}(x) = e \quad \text{env} \vdash e \rightarrow v}{\text{env} \vdash x \rightarrow v}$$

$$(11a) \frac{\text{env} \vdash e1 \rightarrow (\text{fn } x \Rightarrow e3) \quad \text{env}, (x, e2) \vdash e3 \rightarrow v3}{\text{env} \vdash e1 \ e2 \rightarrow v3}$$

2. For static scope, we have:

$$(9b) \frac{\text{env}(x) = (e, \text{env1}) \quad \text{env1} \vdash e \rightarrow v}{\text{env} \vdash x \rightarrow v}$$

$$(10a) \quad \text{env} \vdash (\text{fn } x \Rightarrow e) \rightarrow (\text{fn } x \Rightarrow e, \text{env})$$

$$(11b) \frac{\text{env} \vdash e1 \rightarrow (\text{fn } x \Rightarrow e3, \text{env1}) \quad \text{env1}, (x, (e2, \text{env})) \vdash e3 \rightarrow v3}{\text{env} \vdash e1 \ e2 \rightarrow v3}$$

1. Write an interpreter that implements the call-by-name parameter passing with dynamic scope:

```
interp_name_dynamic: (env * term) -> result
```

2. Write an interpreter that implements the call-by-name parameter passing with static scope:

```
interp_name_static: (env * term) -> result
```

You may need to modify some of your basic datatypes.

## Problem 2: Implementation of lazyness

Your previous interpreter works well if the actual parameter is not invoked. However, if the evaluation of the actual parameter is needed, the actual parameter is evaluated each time the actual parameter is accessed. For example, in the following program:

```
fun f x = x + x + x
f(2+3)
```

the expression

2+3

will be evaluated three times. Write an interpreter that avoids this recomputation and implements static scope. You can make use of the functions `delay` and `force` discussed in class.

## Problem 3: Impact of types on the code

Briefly discuss what you would change in your interpreters if the input to the interpreter had been type-checked before its invocation.

## Problem 4: Delayed evaluation and functions

We can simulate delay evaluation in an eager language such as ML by using higher-order functions.

Consider a sequence data structure that starts with a value and continues with a function (known as a thunk) to compute the rest of the sequence:

```
- datatype 'a Seq = Cons of 'a * (unit -> 'a Seq);

- fun head (Cons (x,_)) = x;
  val head = fn : 'a Seq -> 'a

- fun tail (Cons (_, xs)) = xs();
  val tail = fn : 'a Seq -> 'a Seq
```

This lazy sequence data type provides a way to create infinite sequences, with each infinite sequence represented by a function that computes the next element in the sequence. For example, here is the sequence of infinitely many 1s:

```
- val ones = let fun f () = Cons(1,f) in f() end;
```

We can see how this works by defining a function that gets the *n*th element of a sequence and by looking at some elements of our infinite sequence:

```
- fun get(n,s) = if n = 0 then head s else get(n-1,tail s);
  val get = fn : int * 'a Seq -> 'a

- get(0,ones);
```

```
val it = 1 : int
```

```
- get(5,ones);  
val it = 1 : int
```

```
- get(245, ones);  
val it = 1 : int
```

We can define the infinite sequence of all natural numbers by

```
- val natseq = let fun f n () = Cons(n,f(n+1)) in f 0 () end;
```

Using sequences, we can represent a function as a potentially infinite sequence of ordered pairs. Here are two examples, written as infinite lists instead of as ML code (note that  $\sim$  is a negative sign in ML):

```
add1    = (0, 1) :: (~1, 0) :: (1, 2) :: (~2, ~1) :: (2, 3) :: ...
```

```
double = (0, 0) :: (~1, ~2) :: (1, 2) :: (~2, ~4) :: (2, 4) :: ...
```

Here is ML code that constructs the infinite sequences and tests this representation of functions by applying the sequences of ordered pairs to sample function arguments.

```
- fun make_ints f = let  
    fun make_pos n = Cons((n, f n), fn()=>make_pos(n+1))  
    fun make_neg n = Cons((n, f n), fn()=>make_neg(n-1))  
  in  
    merge (make_pos 0, make_neg (~1))  
  end;  
val make_ints = fn : (int -> 'a) -> (int * 'a) Seq  
  
- val add1 = make_ints (fn x => x + 1);  
val add1 = Cons ((0,1),fn) : (int * int) Seq  
  
- val double = make_ints (fn x => 2*x);  
val double = Cons ((0,0),fn) : (int * int) Seq  
  
- fun apply (Cons( (x1,fx1), xs) , x2) = if (x1 = x2) then fx1 else apply (xs(), x2);  
val apply = fn : ('a * 'b) Seq * 'a->'b  
  
- apply(add1, ~4);  
val it = ~3 : int  
  
- apply(double, 7);  
val it = 14 : int
```

- Write merge in ML. Merge should take two sequences and return a sequence containing the values in the original sequences, as used in the `make_ints` function.
- Using the representation of functions as a potentially infinite sequence of ordered pairs, write compose in ML. Compose should take a function `f` and a function `g` and return a function `h` such that  $h\ x = f\ (g\ x)$ .