Weekly exercises in INF3110
week 41 6-10.10.2008

Exercise 1

Exercise 6.1 in Mitchell's book

a)

```
fun a(x,y) = x+2*y;

val a = fn : int * int -> int
```

All the operations used in the expression are over integers. *a* takes a pair as argument.
(a pair is also a tuple, or more specifically, a 2-tuple)

b)

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

val b = fn : real * real -> real
```

From 2.0 we know that all the operations must be performed over reals. *b* takes a pair as argument.

c)

```
fun c(f)=fn y=>f(y);
```

y is an argument of some type 'a. Since f is applied to y, it must
have function type 'a -> 'b. c takes f as an argument, which is
applied to y, hence the type of c.

```
Example of usage:
c(fn (a,b,c)=>5+a+b+c);  [it = fn : int * int * int -> int ]
c(op +);                 [it = fn : int * int -> int]
c(op +)(2,3);            [it = 5]

val c = fn : ('a -> 'b) -> 'a -> 'b
```

d)

```
fun d(f,x)=f(f(x));

val d = fn : ('a -> 'a) * 'a -> 'a
```

The type of x is some type 'a. f must have a function type
from 'a to 'a (since f is applied to itself which is applied to x).
d takes a pair consisting of the function f (with type 'a->'a) and x:'a.

Example of usage:
```
d(fn(a)=>a+1,5);  [it = 7]
```

e)

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

val e = fn : 'a * 'a * ('a -> bool) -> 'a
```

e takes three arguments. By the definition x and y must have
the same type (restriction on the if-then-else construct in ML),
namely 'a. b must take something of type 'a and return a boolean,
since it is used as a condition.

Example of usage:
```
e(2,3,fn(a)=>a=2); [it = 3]
e(2,3,fn(a)=>a=3); [it = 2]
```

Exercise 2
----

a) Exercise 6.4 in Mitchell's book

```
fun Y f x = f (Y f) x;
val Y = fn : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b

fun F f x = if x=0 then 1 else x*f(x-1);
val F = fn : (int -> int) -> int -> int

val factorial = Y F;
val factorial = fn : int -> int
```

```
Y is a function which takes a function as a first argument, which
applied to an argument of type 'a gives something of type 'b.

Since Y is a fixed-point operator, Y must be applied to a
"functional", like the function F in a.
```

```
Example of usage:
factorial(5);  [it = 120]

Sidenote:

In Scheme, Y, F and factorial is written like this:

(define ((Y f) x)
  ((f (Y f)) x))
(define ((F f) x)
  (if (= x 0)
      1
      ( * x (f (1- x)))))
(define factorial (Y F))
```

b) Apply F to the identity function (fn x => x) and the following arguments: 0, 1, 2, 3. What is supposed to be the result? Why? Is F the factorial function? Explain.

```
- F (fn x => x) 0;
val it = 1 : int
- F (fn x => x) 1;
val it = 0 : int
- F (fn x => x) 2;
val it = 2 : int
- F (fn x => x) 3;
val it = 6 : int
But:
- F (fn x => x) 4;
val it = 12 : int

(0->1, 1->0, 2->2, 3->6, 4->12)

  "F (fn x => x) 0"
```

gives 1 since the first branch of the conditional is applied.

```
  "F (fn x => x) 1"
```

gives 0 since

```
  “1*f(0) = 1*0 = 0
```

(f is the identity function).

F is not a recursive function:

```
    "F (fn x => x) n"
```

gives

```
    “n*(n-1)”
```

.

Exercise 3

Exercise 6.5 in Mitchell's book

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

1. The following shows the assignment of type to nodes:

```
   (lambda) : r
      /   \
     /     \
    /      @ : v
   /      /   \
  /      /     \
 /    @ : u    2 : int
 |    /   \
 |   /     \
 | + : A   @ : t
 |         /  \
 |        /    \
 |____g : tg   h : th
 |______________|


where A =def=  int->int->-int.

2. Constraints
(1) tg = th -> t
(2) int -> (int -> int) = t -> u
(3) u = int -> v
(4) r = (tg * th) -> v

3. Substitution
From (2) we get t = int
            and u = int -> int.

Using this in (3) we get:
                v = int,

and thereby, from t=int,(1) and (4) we get
                r = (th->int) * th -> int.

We substitute with type variables and get:
                r = ('a -> int) * 'a -> int ,

which is the compiler output.
```

Exercise 4

Exercise 6.6 in Mitchell's book

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

1. The following shows the assignment of type to nodes:

```
    (lambda) : r
      /   \
     /     \
    /      @ : v
   /      /   \
  /      /     \
 /    @ : u    2 : int
 |    /   \
 |   /     \
 | + : A   @ : t
 |         /  \
 |        /    \
 |____g : tg   g : tg
 |______________|
```

where A =def= int->int->-int.

```
2. Constraints
(1) tg = tg -> t
(2) int -> (int -> int) = t -> u
(3) u = int -> v
(4) r = tg -> v
```

3. Substitution
Substituting u = int -> v in (2) we get
int -> (int -> int) = t -> (int -> v) from which it follows
that t = int and v = int, and thereby r = tg -> int.

Now, from equation (1) we have that in order to get tg we must know tg, which has no solution. Hence, the type inference algorithm cannot give an appropriate type to the function ("circularity" error). This can also be seen directly by looking at 1.

The compiler outputs:

```
stdIn:4.12-4.20 Error: operator is not a function [circularity]
  operator: 'Z
  in expression:
    g g
```

Exercise 5

Exercise 6.7 in Mitchell's book

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

append: ('a list * 'b) => 'b
```

The first argument must be a list of some type. (since nil is the
empty list, and x::l must be a list). However we cannot infer anything
about the type of the second argument, especially the compiler cannot
infer that the argument m is a list, since no list
operation/constructor is applied to it; that is why m is given type 'b.

If we know that the return type of append is 'b, we know that append
does not always return a list which it should if it was defined correctly.

Actually, if we call append with a list and some value of type 'b, it
just returns the value.

Exercise 6

Given the following general signature specifying a type t
equipped with the value "zero" and operators "sum", "prod", "diff" and "quo":

```
signature ARITH = sig
  type t
  val zero: t
  val sum: t * t -> t
  val diff : t * t -> t
  val prod : t * t -> t
  val quo : t * t -> t
end;
```

Declare a structure "Real" matching signature ARITH, such that Real.t is the type "real" and the
components "zero", "sum", etc, denote the corresponding operations on type "real".

Answer:
```
structure Real : ARITH =
  struct
  type t = real;
  val zero = 0.0;
  fun sum   (x,y) = x+y : t;
  fun diff  (x,y) = x-y : t;
  fun prod  (x,y) = x*y : t;
  fun quo   (x,y) = x/y : t;
  end;

open Real;

sum(2.3,5.9);
```

Exercise 7

There are two kinds of overloading of functions/operators:
- context-independent: overloading only done on parameters to function or type of operands for an operator;
- context-dependent: which operator to use depends also on the type of the result.

Hence, with context-independent overloading, the function to be called is solely based upon the type of the actual parameter. With context-dependent overloading, the function to be called may not be identified by the type of the parameters. The context must be taken into account to identify the function to be called.

a) Consider the operator ++ (concatenation) to be overloaded, with types

```
Char x Char -> String
Char x String -> String
String x Char -> String
String x String -> String
```

Is this overloading context-independent or context-dependent? Have a look at the following examples:

```
c ++ s
(s ++ c) ++ c
s ++ (c ++ c)
```

where c is a Char variable and s is a String variable, and identify for each ++ what kind of concatenation it is.

```
c++s: C*S->S
(s++c)++c: (S*C->S)*C->S
s++(c++c): S*(C*C->S)->S
```

Answer: Context-independent, as all combinations of Char x String are defined.

b) Define in ML four infix operators ++ with the types specified in item a) (in the same order as they are defined). Define then the following four values:

```
val c1 = #"a";
val c2 = #"h";
val s1 = "bcd";
val s2 = "efg";
```

Before executing, guess what would be the result of the following:

```
c1 ++ c2; => "ah"
s1 ++ c2; => "bcdh"
c1 ++ s1; => "abcd"
s1 ++ s2; => "bcdefg"
```

Try it using sml and explain the result.

Answer:

```
infix ++;

fun c ++ s = str(c)^s;
fun c1 ++ c2 = str(c1)^str(c2);
fun s ++ c = s^str(c);
fun s1 ++ s2 = s1^s2;

c1 ++ c2;
s1 ++ c2;
c1 ++ s1;
s1 ++ s2;
```

Exercise 8

(NOTE: This exercise is about overloading in general, not in ML!)
Consider overloading to be defined for proper procedures (that is, not functions).

a) Is it context-dependent or context-independent?

independent.
"since a procedure does not give a result." ???

b) Assume that the operator / is overloaded, with types

```
      int * int -> int

   and

      int * int -> double.
```

Assume further that we have an overloaded proper procedure "write", the argument of which may be either int or double.

Give examples of procedure calls where the procedure to be called cannot be defined uniquely.

```
write(5/9);
```

Exercise 9

Consider the following parametric function in C++ like syntax:

```
template <typename T>
T second(T x, T y) {return y}
```

It is parameterised by the type parameter T, and it returns the
value of the second parameter. It is called in this way:

```
int i,j;
second<int>(i,j)
```

The **<int>** is not really needed, as the compiler may infer that
the type shall be int, but it is included here for clarity reasons.

a) The body of the function does not really use any properties of the type T.
Why is it so that C++ still has to generate different code for different actual types?

Answer:
It will still have to allocate space for x and y as part of the activation record for the call of second, and the space depends upon the type. For example, double usually has size 8, while float has size 4.

b) Assume that we had the following version of "second":

```
template <class T>
T second(T *x, T *y) {return y}
```

Why would it be possible to use the same code for "second" for different actual classes for T?

Answer:
Same argument as above, or rather: pointers will take up the same space independently of the class typing the pointer.