Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Michael Grossniklaus
Claudia Ignat

Informatik 3
Winter Term 2001/2002

Week 3

Exercise 3.1:

1. The following two functions give a definition of `reduce` and `filter` as curried functions. The first function `reduce` is a function that given another function `f` with two arguments, a list and a initial value, uses the given function to aggregate all values of the list.

```
fun reduce f nil a = a
|   reduce f (hd::tl) a = f hd (reduce f tl a);

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

   The second function `filter` takes a function `f` and a list as arguments. It then uses the given function to filter the list. Any element of the list for which `f` returns `true` will be included in the result list.

```
fun filter f nil = nil
|   filter f (hd::tl) = if (f hd) then hd::filter f tl
                        else filter f tl;

val filter = fn : ('a -> bool) -> 'a list -> 'a list
```

2. To find the length of the shortest string in a list of strings we use function `Integer.min` instead of `Integer.max` as in the example given in the lecture notes. The problem however is what initial value we should give to the `reduce` function. SML/NJ '97 offers a function `Integer.maxInt` that returns an `int option` containing the largest possible integer value of the system. Hence we use this value to initialize the `reduce` function.
   But first it is necessary to define a helper function `minSize`. This is necessary since function `Int.min` cannot be applied as a curried function and is therefore useless in the context of a curried `reduce` as defined above.

```
fun minSize x a = Int.min(String.size x, a);

val minSize = fn : string -> int -> int
```

   The following is the final function that finds the length of the smallest string in the given list. In our example this is of course `3`, the length of the string `SML`. The function `Option.valOf` is necessary to convert the `int option` returned by `Int.maxInt` into an `int` value.

```
reduce minSize ["Java", "Oberon", "SML"] (Option.valOf Int.maxInt);

val it = 3 : int
```

3. The function `avlength` given below returns the average length of strings given as arguments as a list of strings. Again we have to define a helper function that allows application in curried form. This time we have to define a curried function `sum` to aggregate the total length of strings in the given list. Unfortunately it is not possible to use `op +` as it not defined as a curried function.

   ```
   fun sum x y = x + y;

   val sum = fn : int -> int -> int
   ```

   Now we can use `map` to convert the list of strings into a list of integers representing the lengths of the strings. Using `reduce` we can sum these values to recieve the total length of all strings in the list. Finally this sum has to be divided by the number of strings contained in the list.

   ```
   fun avlength nil = 0.0
   |   avlength slist = real(reduce sum (map String.size slist) 0)
                      / real(length slist);

   val avlength = fn : string list -> real
   ```

4. To solve this exercise we modify our definitions of `map` and `reduce` to work on binary trees that comply to the definition given in the exercise.

   (a) The following function `mapTree` performs a map operation over a binary tree.

   ```
   fun mapTree f empty = empty
   |   mapTree f (node(x, left, right)) =
                  node(f x, (mapTree f left), (mapTree f right));

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

   To show how `mapTree` could be used to transform `myFamilyTree` into a binary tree of the same structure but with the length of the names of the family members as the node labels, we use function `String.size` and map it to every node in the tree. To verify if everything is correct we shall apply our function to the following example.

   ```
   val myFamilyTree =
     node("Paul", node("Peter", node("John", empty, empty),
                                node("Mary", empty, empty)),
                  node("Elizabeth", node("Fred", empty, empty),
                                    node("Victoria", empty, empty)));

   val myFamilyTree =
     node ("Paul",node ("Peter",node #,node #),node ("Elizabeth",node #,node #))
       : string tree

   mapTree String.size myFamilyTree;

   val it = node (4,node (5,node #,node #),node (9,node #,node #)) : int tree
   ```

   (b) Function `redTree` given below implements a reduce over a binary tree. As function `f` is a so-called aggregation function it is important that it takes exactly two arguments, even if an extension to three or more arguments would simplify the definition of `redTree`.

   ```
   fun redTree f a empty = a
   |   redTree f a (node(x,left,right)) = redTree f (redTree f (f a x) left) right;

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

Next we want to define a function `treeToString` that transforms a family tree into a string of names of the family members prefixed with "my family members : " and separated by spaces. To be able to use the `reduce` function we first have to define an aggregation function `join` that concatenates to strings with a blank between them.

```
fun join s1 s2 = s1 ^ " " ^ s2;

val join = fn : string -> string -> string
```

Now we can define function `treeToString` using `reduce` and `join` in the following way. **Note:** Instead of defining a function using the `fun` keyword, we use the `val` syntax as this allows us to omit the list parameter of the `reduce` function.

```
val treeToString = redTree join "my family members : ";

val treeToString = fn : string tree -> string

treeToString myFamilyTree;

val it = "my family members : Paul Peter John Mary Elizabeth Fred Victoria"
  : string
```

(c) The type of `myFun` can be determined using the SML system.

```
val myFun = mapTree treeToString;

val myFun = fn : string tree tree -> string tree
```

5. The following is a possible definition of a function `iter` that complies to the definition given in the tutorial.

```
fun iter 0 f x = x
|   iter n f x = iter (n-1) f (f x);
```

The type of this function is easily determined. The function takes three arguments `'a -> 'b -> 'c`. From the first match condition and left-hand side of the second clause, we can see, that `'a` has to be type `int`. The second argument `f` has to be a function which takes values of type `'c` (the type of `x`) and returns values of type `'c`, hence the second argument is of type `('c -> 'c)`. The third argument `x` as mentioned before is simply a value of type `'c`. From the first clause of the function it becomes clear that the result of `iter` has to be of the same type as `x`, which means the function returns values of type `'c`. The complete type string therefore is `int -> ('c -> 'c) -> 'c -> 'c`. Below is the type as SML would compute it, which is equivalent with respect to renaming.

```
val iter = fn : int -> ('a -> 'a) -> 'a -> 'a
```

To compute x^4 we can define the following function.

```
fun power4 x = iter 2 (fn x => x * x) x;

val power4 = fn : int -> int
```

6. To determine which one of the given two expressions is correct we attempt to evaluate their type. If we're able to find a type expression the example is correct.

- `compose(compose, uncurry compose)`
  The first step when trying to find the type of an expression is always to check if the expression can be evaluated, or if there are missing arguments that prevent us from applying the definitions of the occurring functions.

In our example we cannot evaluate the outermost `compose` as there is a missing argument. Therefore the next step is to add arguments to the expression until evaluation of the expression becomes possible. In our case here, it is enough to just add one more argument `x`.

```
     compose(compose, uncurry compose) x
 =   compose(uncurry compose x)
```

Again evaluation is impossible, as there is no second argument for the outermost compose. Hence we add another argument `y` to the original expression. Further we can see that the term `(uncurry compose x)` has to be of the form `(f, g)` or else evaluation is not possible.

```
     compose(compose, uncurry compose) x y
 =   compose(uncurry compose x) y
 =   compose(f, g) y                                    [1]
 =   f(g y)                                             [2]
```

As the type of a function is of the form *input* → *output* we now can establish the general form of the type of the given expression.

$$\tau(x) \to \tau(y) \to \tau(f(g\ y))$$

Note: $\tau(x)$ has to be read as "the type-string of x" and is no mathematical function but rather a textual replacement.

Having come this far, we now need to concentrate on the type of `x`, `y` and `f(g x))`. To do so, we have another look at the term `uncurry compose x`.

	`uncurry compose x`	*to apply* `uncurry`*,* `x` *has to be of form* `(u, v)`
=	`uncurry compose (u, v)`	
=	`compose u v`	*to apply* `compose`*,* `u` *has to be of form* `(i, j)`
=	`compose (i, j) v`	
=	`i(j v)`	[3]

With this extended knowledge about the structure of the arguments of the given expression, it is possible to write down a correct invocation of `compose(compose, uncurry compose)`.

```
compose(compose, uncurry compose) ((i, j), v) y
```

But this is not the only progress we've made. We now have an improved form of the type-string given further above.

$$\tau(((i,j),v)) \to \tau(y) \to \tau(f(g\ y))$$

The final step is to simplify this type-string by substituting the τ by types. To do so, we have to find the arguments that are simply values and assign them a type. If we cannot find out if the type should be a base-type like `int`, `real` or `string`, we simply assign ascending type variables (`'a`, `'b`, `'c`, ...).

- From [2] it follows that `y` is a value. The type can be chosen to be `'a`.
- From [3] it follows that `v` is a value. The type can be chosen to be `'b`.
- From [3] it follows that `j` is a function that takes values of type `'b` as input. The output type is not inferable and can be chosen to be `'c`. It follows that `i` is of type `('b -> 'c)`.
- From [3] it follows that `i` is a function that takes values of type `'c` (the output type of function `j`) as input.

  From [3] it follows that the output of `i` is of the same type as `(uncurry compose x)`. In [1] we noted that the term `(uncurry compose x)` is of form `(f, g)`, hence the result of `i` has to be a tuple of functions.

  - The inner function `g` is applied to `y` which is of type `'a`. Its output is not inferable, but has to match the input of the outer function as `f` is applied to the result of `g`. We choose `'d` as output of `g` and input of `f`. It follows, that the type of `g` is `('a -> 'd)`.
  - The outer function `f` as mentioned before has input type `'d`. The output type is not inferable, so we choose `'e` and get `('d -> 'e)` as the type for `f`.

  Putting all this together it follows that the type of `i` has to be `'c -> ('d -> 'e) * ('a -> 'd)`.

Now we know the type of every variable in the type-string given above. Hence we can substitute the τ for the results we have found.

```
(('c -> ('d -> 'e) * ('a -> 'd)) * ('b -> 'c)) * 'b -> 'a -> 'e
```

SML of course would perform a final renaming of the types in alphabetical order from left-to-right. With this renaming the resulting type-string is as follows.

```
(('a -> ('b -> 'c) * ('d -> 'b)) * ('e -> 'a)) * 'e -> 'd -> 'c
```

- `compose(uncurry compose, compose)`
  We proceed here as in the example before and think about the arguments that we would need to add in order to evaluate the expression.

	`compose(uncurry compose, compose) x`	*to evaluate* `compose`*, we assume* `x`	[1]
=	`uncurry compose(compose x)`	`(compose x)` *has to be of form* `(g, h)`	[2]
=	`uncurry compose(g, h)`		
=	`compose g h`	it `g` has to be of form `(i, j)`	
=	`compose (i, j) h`		
=	`i(j h)`		

  From this we can see, that `x` is of the form `((i, j), h)`. Giving an argument of this form however is not possible in this particular example!

  On the one hand, we need to put the outer brackets of the argument, in order to have it recognized as **one** value by the system at [1]. On the other hand these brackets prevent a correct evaluation of the innermost `compose` at [2] where **two** values are expected. Therefore this function cannot be evaluated correctly.

7. Before we try to evaluate the type of `app app` we have to find the type of function `app` itself. Function `app` has two arguments `g` and `f` whose types we try to infer first.

   - From the left-hand side it is clear, that `g` is a value. The type of `g` is not inferable and can be chosen to be `'a`.
   - Argument `f` is a function which takes values of the same type as `g` as input. As `f` is applied to itself, input and output have to be of the same type. It follows, that the type of `f` is `('a -> 'a)`.
   - The result type of function `app` is the same as the result type of function `f` which we found to be `'a`.

   The whole type-string of `app` therefore is `'a -> ('a -> 'a) -> 'a`.

   Having established this, we proceed to the evaluation of the expression `app app`. Again we use the method introduced in the previous exercise.

	`app app x`	*to evaluate the outermost* `app`*,* `x` *has to be added*	
=	`x(x app)`		[1]

   Having evaluated the expression this far, it is possible to write down the following general form of the type-string that function `app app` is going to have.

$$\tau(x) \rightarrow \tau(x(x\ app))$$

   Knowing this it is reasonable to think about the type of `x` as it is the only remaining unknown type in the above type-string.

   - From [1] it follows that `x` is a function. As `x` is applied to itself it has to be of form `('b -> 'b)`.
   - From [1] it follows that the input of `x` is of the same type as function `app`. The type of `x` therefore is `('a -> ('a -> 'a) -> 'a) -> ('a -> ('a -> 'a) -> 'a)`.

   Finally we can substitute all τ and get the following type-string. No renaming has to be preformed as there is only one type variable occuring in the type-string.

```
(('a -> ('a -> 'a) -> 'a) -> ('a -> ('a -> 'a) -> 'a)) -> ('a -> ('a -> 'a) -> 'a)
```

Exercise 3.2:

1. The function `insert` given below performes an insert into an ordered integer binary tree. If a node with the given label already exists in the binary tree the function does not change the tree.

```
fun insert(x:int,empty) = node(x,empty,empty)
|   insert(x:int,node(y,left,right)) =
          if x = y then node(x,left,right)
                   else if x > y then node(y,left,insert(x,right))
                                 else node(y,insert(x,left),right);

val insert = fn : int * int btree -> int btree
```

2. The following function `lookup` checks if a given value exists as label of a node inside the tree. If so, the function returns `true` and `false` otherwise.

```
fun lookup(x:int,empty) = false
|   lookup(x:int,node(y,left,right)) =
          if x = y then true
                   else if x > y then lookup(x,right)
                                 else lookup(x,left);

val lookup = fn : int * int btree -> bool
```

3. Deletion inside ordered binary trees is a bit tricky, as there has to be a suitable replacement for the deleted node. The following functions give a possible solution in SML.

```
fun merge(empty,T) = T
|   merge(T,empty) = T
|   merge(node(x,left,right),T) = node(x,left,merge(right,T));

val merge = fn : 'a btree * 'a btree -> 'a btree

fun delete(x,empty) = empty
|   delete(x,node(y,left,right)) =
           if x = y then merge(left,right)
                    else if x < y then node(y,delete(x,left),right)
                                  else node(y,left,delete(x,right));

val delete = fn : int * int btree -> int btree
```