Programming Language Concepts, 2003 Semester 1

Tutorial 06 and 07, CS2104

1 Making `Length` Tail-Recursive

One of our running examples has been `Length` which computes the length of a given list. We start from the following function:

```
fun {Length Xs}
   case Xs
   of nil  then 0
   [] X|Xr then 1+{Length Xr}
   end
end
```

The goal is to derive a tail-recursive implementation of `Length` by using the same steps as in the lecture:

1. Sketch how `{Length [a b c]}` computes.
2. Rearrange the additions `1+`$\cdots$.
3. Find a state invariant.
4. Give a tail-recursive implementation.
5. Convince yourself that the state invariant is maintained by your implementation.
6. Give a complete definition of `Length` that uses tight scope.

Solution. See book, Section 3.4.2.

2 Computing Binary Logarithms

The binary logarithm l of an integer n with $n > 0$ is the unique number that satisfies

$$2^l \leq n < 2^{l+1}$$

According to this definition, the binary logarithm of 3 is 1, of 4 is 2, and of 5 is 2.
We can compute the binary logarithm by successively trying increasing values for l until the above relation is satisfied.

Give a tail-recursive implementation `Ld` of the binary logarithm function. Use an auxiliary function to find the right value for l. Note that you don't need the power function!

Solution. The idea behind our algorithm is to have two accumulators: one for l and one for 2^l. The recursive procedure will always maintain the relation between these two numbers and will continue until 2^{l+1} is larger than n.

```
local
   fun {Up L TwoL N}
      if 2*TwoL>N then L
      else {Up L+1 2*TwoL N}
      end
   end
in
   fun {Ld N}
      {Up 0 1 N}
   end
end
```

3 Making Computing the Fibonacci Numbers Fast

A frequently occurring sequence of numbers are the Fibonacci numbers, which are defined inductively as follows (for $n \geq 0$):

$$f(n) = \begin{cases} 0 & \text{, if } n = 0 \\ 1 & \text{, if } n = 1 \\ f(n-2) + f(n-1) & \text{, otherwise} \end{cases}$$

1. Give a recursive implementation `Fib` for the function f.
2. Give a tail-recursive implementation of `Fib`.

   Hint: Use two accumulators for the two previous Fibonacci numbers.

Solution.

1. Here we go:

```
fun {Fib N}
   case N
   of 0 then 0
   [] 1 then 1
   else {Fib N-2}+{Fib N-1}
   end
end
```

2. The function uses two accumulators `F2` and `F1` for maintaining the $(i-2)$-th and $(i-1)$-th Fibonacci number.

```
local
   fun {FibUp F2 F1 I}
      if I==0 then F2+F1
      else {FibUp F1 F2+F1 I-1}
      end
   end
in
   fun {Fib N}
      case N
      of 0 then 0
      [] 1 then 1
      else {FibUp 0 1 N-2}
      end
   end
end
```

4 Producing Power Functions

Consider the following variant of the `Pow`-function discussed in Lecture 06:

```
fun {PowFactory X}
   fun {PowAcc N A}
      if N==0 then A
      else {PowAcc N-1 X*A}
      end
   end
   fun {Pow N}
      {PowAcc N 1}
   end
in
   Pow
end
```

What is returned by this function? Try the following examples to find out:

```
declare P2={PowFactory 2}
declare P5={PowFactory 5}
{Browse {P2 4}}
{Browse {P2 7}}
{Browse {P5 3}}
{Browse {P5 6}}
```

Give the external references for `PowAcc` and `Pow`. Give the contextual environments for `PowAcc` and `Pow` in the above examples.

Solution. The external references for `PowAcc` are: `PowAcc` itself and `X`. The external reference for `Pow` is `PowAcc`.

The contextual environment for `PowAcc` is $\{\texttt{PowAcc} \mapsto pa, \texttt{X} \mapsto x\}$ where pa is the store variable that refers to the procedure value for `PowAcc` and x is the store value for the variable identifier `X` as defined by the environment used by execution of the procedure body of `PowFactory`.
The contextual environment for `Pow` is $\{\texttt{Pow} \mapsto p, \texttt{PowAcc} \mapsto pa\}$ where pa is as above and p is the store variable refereing to the procedure value for `Pow`.

5 Filtering List Elements

Implement a function `{Filter Xs F}` that returns all elements of `Xs` in order for which `F` returns `true`.

Solution.

```
fun {Filter Xs F}
   case Xs
   of nil then nil
   [] X|Xr then
      if {F X} then X|{Filter Xr F}
      else {Filter Xr F}
      end
   end
end
```

6 Left and Right Folding

Try the following examples to better understand `FoldL` and `FoldR`:

1. `{Browse {FoldL [a b c d] Snoc nil}}`
2. `{Browse {FoldR [a b c d] Cons nil}}`
3. `{Browse {FoldL [a b c d] Cons nil}}`
4. `{Browse {FoldR [a b c d] Snoc nil}}`

with the following definitions

```
fun {Cons X Xr} X|Xr end
fun {Snoc Xr X} X|Xr end
```

7 Computing Maximum with Fold

Compute the maximum element from a list of numbers by folding. What is the initial value to choose for passing to `FoldL` or `FoldR` (remember: there is no smallest integer as integer precision is unlimited)?
Which version of folding are you using (`FoldL` or `FoldR`)? Why?

Solution. Both are possible and compute the same result. However, one should use `FoldL` as it is tail-recursive as opposed to `FoldR`.
A solution is as follows

```
fun {MaxList Xs}
   {FoldL Xs.2 Max Xs.1}
end
```

8 Rewriting Combinations of `Map` and `FoldL`

Your friend shows you the following program fragment:

```
Ys={FoldL {Map Xs F} G S}
```

Can you give a program fragment that computes the same result but does only use `FoldL`? Does the same trick work for `FoldR`?

Solution. The idea is to apply `F` before `G` is applied by `FoldL`:

```
Ys={FoldL Xs fun {$ S X}
                {G S {F X}}
             end S}
```

It is analogous for `FoldR` (beware of the order of arguments for `G`).

Your friend is really pushing it, she wants you to write a function `{FoldMap F G S}` that returns a function that takes a list `Xs` and returns a list computed according to the above code fragment. Try to please your friend!

Solution.

```
fun {FoldMap F G S}
   fun {FG S X}
      {G S {F X}}
   end
in
   fun {$ Xs}
      {FoldL Xs FG S}
   end
end
```

It is analogous for `FoldR` (beware of the order of arguments for `G`).

9 Testing List Elements

Develop functions `{All Xs F}` and `{Some Xs F}`, where `Some` tests whether there exists an element in `Xs` for which `F` returns `true` and `All` tests whether `F` returns `true` for all elements in `Xs`.

Solution.

```
fun {All Xs F}
   case Xs
   of nil then true
   [] X|Xr then
      if {F X} then {All Xr F} else false end
   end
end
fun {Some Xs F}
   case Xs
   of nil then false
   [] X|Xr then
      if {F X} then true else {Some Xr F} end
   end
end
```

10 Mapping Tuples

Develop a function `{MapTuple T F}` that returns a tuple that has the same width and label as the tuple `T` with its fields mapped by the function `F`.
For example,

```
local
   fun {Sq X} X*X end
in
   {MapTuple a(1 2 3) Sq}
end
```

should return `a(1 4 9)`.
Remember, a tuple is constructed with `{MakeTuple L N}`, where `L` is the label and `N` is the width of the tuple.

Solution. Note that the mapping takes place in order from last field to first field.

```
local
   proc {Map I T1 T2 F}
      if I>0 then
         T2.I={F T1.I} {Map I-1 T1 T2 F}
      end
   end
in
```

```
   fun {MapTuple T1 F}
      N ={Width T}
      T2={MakeTuple {Label T} N}
   in
      {Map N T1 T2 F}
      T2
   end
end
```

11 Merging Lists

Suppose you have two sorted lists. Merging is a simple method to obtain an again sorted list containing the elements from both lists. The following program gives merging for a list of numbers:

```
fun {MergeNum Xs Ys}
   case Xs
   of nil  then Ys
   [] X|Xr then
      case Ys
      of nil then Xs
      [] X|Xr then
         if X<Y then X|{MergeNum Xr Ys}
         else Y|{MergeNum Xs Yr}
      end
   end
end
```

Enhance the above program to a `Merge` function that is generic with respect to the order (that is, it takes a binary function as argument that tests whether its first argument is smaller than its second).

Solution.

```
fun {Merge Xs Ys F}
   case Xs
   of nil  then Ys
   [] X|Xr then
      case Ys
      of nil then Xs
      [] X|Xr then
         if {F X Y} then X|{Merge Xr Ys F}
         else Y|{Merge Xs Yr F}
      end
   end
end
```

12 MergeSort

Take the above `Merge`-function to implement sorting by merging (short: `MergeSort`) as follows:

- An empty list is sorted.
- A list with one element is sorted.
- Otherwise, split a non empty list in two lists of approximately equal length (say all elements at odd position go to one list, the other elements go to the other list). Write a function `Split` implementing this idea.
- Sort the lists obtained from splitting by recursively applying `MergeSort`.
- Merge the two sorted lists.

Solution.

```
local
   proc {Split Xs As Bs}
      case Xs
      of nil then As=nil Bs=nil
      [] X|Xr then Ar in
         As=X|Ar {Split Xr Bs Ar}
      end
   end
in
   fun {MergeSort Xs F}
      case Xs
      of nil then nil
      [] [X] then [X]
      else As Bs in
         {Split Xs As Bs}
         {Merge {MergeSort As F} {MergeSort Bs F} F}
      end
   end
end
```

13 Trees

Go through the section in the book for trees. You should cover ordered binary trees having the following type:

```
<BTree> ::= leaf
          |  tree(key:<Literal> value:<Value>
                  left:<BTree> right:<BTree>)
```

An Ordered binary tree `T` is a tree such that the keys of the left substree `LT` are less than the key of the root, which in turn is less than the keys of the right subtree. Also the left and right subtrees are ordered. Define the following functions:

- `MakeTree` returns an empty tree, i.e. `leaf`.
- `Insert` has the type `<fun{$ <BTree> <Literal> <Value>}:<BTree>>`. It takes a tree, a literal `K` and any value `X` and returns a tree where node with `K` and `X` has been added.
- `Remove` has the type `<fun{$ <BTree> <Literal>}:<BTree>>`. It takes a tree, a literal `K`, and returns a tree where the node with key `K` has been removed if it exists, otherwise the tree is unchanged.
- `FindCond` has the type `<fun{$ <BTree> <Literal> <Value>}:<Value>>`. It takes a tree, a key, and a default value. If it finds a node in the tree with the key it returns the corresponding value, otherwise it returns the default value.

14 Abstract Data Types

Implement the dictionary ADT using the tree representation in the previou excercise. Implement the ADT in a secure way using the `Wrappen` and `UnWrapper` abstractions. We assume the type of a dictionary is called `<DICT>`. The ADT has the following functions:

- `<fun{NewDictionary}:<DICT>>`: takes no arguments returns an empty dictionary.
- `<fun{Put <DICT> <Literal> <Value> }:<DICT>>`: adds an element to the dictionary.
- `<fun{Delete <DICT> <Literal>}:<DICT>>`: removes an element from the dictionary.
- `<fun{GetCond <DICT> <Literal> <Value>}:<Value>>`: gets an item from the dictionary. if it is not found it returns a default value.