Programmeringsspråk TDT4165 - Assignment 3

Ola Kristoffer Hoff

3. October 2022

Procedural Abstraction

1 Task 1

A Implement proc {QuadraticEquation A B C ?RealSol ?X1 ?X2}. X1 and X2 should bind to the real solution(s) to the quadratic equation. RealSol binds to true if there exists a real solution, false otherwise. See the appendix for more information about the quadratic equation. When there are no real solution, you may ignore complex solutions, and just set RealSol to false.

```
3  % 1.a
4  declare QuadraticEquation
5  proc {QuadraticEquation A B C ?RealSol ?X1 ?X2}
6     UnderRot = (B * B) - 4.0 * A * C in
7     if UnderRot < 0.0 then
8        RealSol = false
9     else
10       RealSol = true
11    end
12    X1 = (~B + {Float.sqrt UnderRot}) / (2.0 * A)
13    X2 = (~B - {Float.sqrt UnderRot}) / (2.0 * A)
14 end
```

Figure 1: Code from 1a.

First I calculate the value under the square root. Then I use this value to determine if RealSol is true or false. If we have a negative value we get the square root of a negative number which is a complex number, hens not real. X1 and X2 don't need to be calculated unless they are real, but this way they get the value "~na" when complex.

B Use your implementation of QuadraticEquation and System.show to answer the following questions:

What are the values of X1, X2, and RealSol, when A = 2, B = 1 and C = -1?

What are the values of X1, X2, and RealSol, whenA = 2, B = 1 and C = 2?

```
42 local RealSol X1 X2 in
43    {System.showInfo "Task 1b: input: A: 2 B: 1 C: -1"}
44
45    {QuadraticEquation 2.0 1.0 ~1.0 RealSol X1 X2}
46    {System.showInfo "RealSol: "}
47    {System.show RealSol}
48    {System.showInfo "X1: "}
49    {System.show X1}
50    {System.showInfo "X2: "}
51    {System.show X2}
52 end
53 local RealSol X1 X2 in
54    {System.showInfo "Task 1b: input: A: 2 B: 1 C: 2"}
55
56    {QuadraticEquation 2.0 1.0 2.0 RealSol X1 X2}
57    {System.showInfo "RealSol: "}
58    {System.show RealSol}
59    {System.showInfo "X1: "}
60    {System.show X1}
61    {System.showInfo "X2: "}
62    {System.show X2}
63
64 end
```

Figure 2: Code for printing the results in 1b.

```
Task 1b: input: A: 2 B: 1 C: -1
RealSol:
true
X1:
0.5
X2:
~1
Task 1b: input: A: 2 B: 1 C: 2
RealSol:
false
X1:
~an
X2:
~an
```

Figure 3: The output of 1b.

Here in figure 2 you can see the code for how I used the procedure and calcualted the values for the two given scenarios. Figure 3 shows the ouput values calculated.

C Why are procedural abstractions useful?

They are very helpfull to reduce code repetition. If we want to do the same thing multiple places we can simply do it once (code it once to be more spesific) in the procedure, then we can use the procedure later when needed.

D What is the difference between a procedure and a function?

The difference between a function and a procedure is quite simple. A function returns a value, while a procedure don't. One could always send an extra variable into a procedure and use that as the "returned" value. This is essentially what happens with a function. So a function is just some syntactic sugar for the user.

Genericity

2 Task 2

```
% Task 2

declare Sum
fun {Sum List}
   case List of Head | Tail then
      Head + {Sum Tail}
   else
      0
   end
end

{System.showInfo "Task 2"}


{System.showInfo "Task 2: input: [1 2 3 4 5 6]"}

{System.show {Sum [1 2 3 4 5 6]}}
```

Figure 4: Code from task 2.

In figure 4 we can see the code used to get the sum of a list.

3 Task 3

```
84 % Task 3
85
86 % 3ab
87 declare RightFold
88 fun {RightFold List Op U}
89    case List of Head | Tail then
90       {Op Head {RightFold Tail Op U}}
91    else
92       U
93    end
94 end
95
96 {System.showInfo "Task 3"}
97
98 {System.showInfo "Task 3a: input: SUM [1 2 3 4 5 6]"}
99 % 3c
100 {System.show {RightFold [1 2 3 4 5 6] fun {$ A B} A + B end 0}}
101
102 {System.showInfo "Task 3a: input: LENGTH [1 2 3 4 5 6]"}
103 %3c
104 {System.show {RightFold [1 2 3 4 5 6] fun {$ A B} 1 + B end 0}}
```

Figure 5: The code for task 3.

A Implement fun {RightFold List Op U}

We can see the implementation figure 5

B Explain each line of code in RightFold in your own words.

I will refer to the line numbers in figure 5.

87: Declare the function *RightFold* to be available globally.
88: Declare the actual function header given in the task.

89: Do a case operation on the List. If it can be split in a *Head* and *Tail* part we go to line 90, otherwise I go to line 92.
90: I call the anonymous Op function with the Head and the recursive of the *RightFold* with the *Tail* part of the *List*.
91: Just the else clause in the case-statment.
92: Here the netural value *U* is returned as a base case.
93: *End* for the case-statment.
94: *End* for the fun-statment.

C Implement fun {Length List} and {Sum List} using RightFold.

At the bottom half of figure 5 we can see the implementation of *Length* and *Sum* with *RightFold*. I call the *RightFold*-function with a list, an anonymous function to specify the behaviour of *Length* and *Sum*, an finally the neutral element U as zero in both cases.

D For the Sum and Lenght operations, would LeftFold (a left-associative fold) and RightFold give different results? Can you provide an example of an operation for which the two folds do not produce the same result?

No, a left-associative fold would not give *Length* nor *Sum* different answers. It does not matter in which order one count the number of elements as long as every element is counted and only counted once. Also the sum will be the same since $a + b = b + a$ mathematically, so order of execution has no effect.

This is not the case for all operations though. One operation that is "fold-sensitive" is division. $\frac{1}{2} \neq \frac{2}{1}$; as we can see mathematically it matters which order we do the operation with division.

E What is an appropriate value for U when using RightFold to implement the product of list elements?

When implementing the product we would multiply all the elements together so to have the base case not effect the answer it would be one. Since $1 * x = x$.

Instantiation

4 Task 4

```
108 % Task 4
109 declare Quadratic
110 fun {Quadratic A B C}
111    fun {$ X}
112       A * X * X + B * X + C
113    end
114 end
115
116 {System.showInfo "Task 4"}
117
118
119 {System.showInfo "Task 4: input: {{Quadratic 3 2 1} 2}"}
120
121 {System.show {{Quadratic 3 2 1} 2}}
```

Figure 6: Code for task 4.

In figure 6 we can see the code for task 4.

Embedding

5 Task 5

A Implement fun {LazyNumberGenerator StartValue} that generates an infinite list of incrementing integers on demand, using higher-order programming. Do not use the built-in Lazy keyword neither take advantage of threads or dataflow variables to implement the laziness.

Here we can see, in figure 7, the code used to solve the task.

```
124 % Task 5
125 % 5a
126 declare LazyNumberGenerator
127 fun {LazyNumberGenerator StartValue}
128    StartValue | fun {$} {LazyNumberGenerator StartValue + 1} end
129 end
130
131 {System.showInfo "Task 5"}
132
133
134 {System.showInfo "Task 5a: calls: {LazyNumberGenerator 0}.1"}
135
136 {System.show {LazyNumberGenerator 0}.1}
137
138 {System.showInfo "Task 5a: calls: {{LazyNumberGenerator 0}.2}.1"}
139
140 {System.show {{LazyNumberGenerator 0}.2}.1}
141
142 {System.showInfo "Task 5a: calls: {{{{{LazyNumberGenerator 0}.2}.2}.2}.2}.2}.1"}
143
144 {System.show {{{{{LazyNumberGenerator 0}.2}.2}.2}.2}.2}.1}
```

Figure 7: Code for task 5a.

B Give a high-level desciption of your solution and point out any limitations you find relevant.

All I did was to return a list with two elements. The head was the *StartValue* and the tail was an anonymous-function that when called would call the *LazyNumberGenerator*-function with the *StartValue* incremented. This anonymous-function will be called when accessing the second element (the tail) in the list object.

Tail Recursion

6 Task 6

A Is your Sum function from Task 2 tail recursive? If yes, explain why. If not, implement a tail-recursive version and explain which changes you needed to introduce to make it tail recursive.

My implementation of the *Sum*-function is tail recursive. It is so because I use the tail part of the list element to do the recursive call. I other words I substitute the list input with it's own tail when called recursive.

B What is the benefit of tail recursion in Oz?

The benefit of tail recursion in *Oz* is that the lists are built up as a pair of a head and a tail, where the tail element refers to another head and tail pair (or nil to indicate the end of the list). So the list object is not a continuous sequence of elements, but resemble or a linked-list data structure. So one cannot iterate over the elements as one would in other languages, so the tail recursion lends itself nicly to travese the list object.

C Do all programming languages that allow recursion benefit from tail recursion? Why/why not?

Absolutely not. If a problem can be solved with recursion or an iterative approach one should almost always use the iterative approach. Recursion uses a lot more memory an should not be used unless necessary. It is a very powerful tool, but it can create inefficient programs and can also cause the programs to crash.