TDT4165 - Programming Languages

Assignment 3: Higher-Order Programming

Benjamin Zubača

October 2, 2024

Task 1

a)

```
proc {QuadraticEquation A B C}
   RealSol
   X1
   X2
   Discriminant = B*B - 4.0*A*C
in
   if Discriminant >= 0.0 then
      RealSol = true
      X1 = (~B + {Float.sqrt Discriminant}) / (2.0*A)
      X2 = (~B - {Float.sqrt Discriminant}) / (2.0*A)
   else
      RealSol = false
      X1 = X2 = noSol
   end
end
```

The procedure `{QuadraticEquation A B C ?RealSol ?X1 ?X2}` takes three parameters `A`, `B`, and `C`, which correspond to the coefficients in the quadratic equation $Ax^2+Bx+C = 0$. It calculates the discriminant $B^2 - 4AC$ to determine the nature of the solutions. If the discriminant is non-negative, it binds `RealSol` to `true` and computes real roots `X1` and `X2` using the quadratic formula. Otherwise, it binds `RealSol` to `false` and both `X1` and `X2` to the atom `noSol`, indicating that no real solutions exist.

b) When `A = 2`, `B = 2`, and `C = ~1`, then:

- `RealSol = true`
- `X1 = 0.5`
- `X2 = ~0.5`

When `A = 2`, `B = 2`, and `C = 2`, then:

- `RealSol = false`
- `X1 = noSol`
- `X2 = noSol`

c) Procedural abstractions are useful because they allow you to encapsulate a sequence of related statements into a reusable procedure. By abstracting these operations into a single unit, the procedure can be invoked with different arguments, simplifying code, enhancing clarity, and facilitating code reuse.

d) A procedure is a syntactic construct that executes a series of statements, performing operations without producing a return value. A function, on the other hand, executes statements (if applicable) *and* returns the value of an expression based on its inputs.

Task 2

```
fun {Sum List}
   case List
   of nil then 0
   [] Head|Tail then Head + {Sum Tail}
   end
end
```

Task 3

a)

```
1 fun {RightFold List Op U}
2    case List
3    of nil then U
4    [] Head|Tail then {Op Head {RightFold Tail Op U}}
5    end
6 end
```

b) Line 1 defines the function `RightFold` with the three parameters `List`, `Op`, and `U`. Line 2 starts a case expression to pattern match on the input `List`. It checks the structure of the `List` to decide the next steps. Line 3 handles the base case: If `List` is empty, it returns `U`, which is the neutral element. Line 4 handles the recursive case: If `List` has `Head` element followed by `Tail` list, it applies `Op` to `Head` and the result of recursively folding the `Tail` using `RightFold`. Lines 5 and 6 indicate the end of the case expression and the `RightFold` function, respectively.

c)

```
fun {Length List}
   {RightFold List fun {$ _ Y} Y + 1 end 0}
end
```

```
fun {Sum List}
   {RightFold List fun {$ X Y} X + Y end 0}
end
```

d) `LeftFold` and `RightFold` would not give different results for the `Sum` and `Length` operations, as they are both associative and commutative.

e) For implementing the product of list elements using `RightFold`, the appropriate value for `U` would be 1. This is because the neutral element for multiplication is 1, as any number multiplied by 1 remains unchanged.

Task 4

```
fun {Quadratic A B C}
   fun {$ X}
      A*X*X + B*X + C
   end
end
```

The function `{Quadratic A B C}` takes three parameters corresponding to the coefficients of the quadratic polynomial $Ax^2 + Bx + C$. It returns an anonymous function that accepts one parameter `X` and calculates the polynomial's value, using the coefficients that are already bound from the outer `Quadratic` function.

Task 5

a)

```
fun {LazyNumberGenerator StartValue}
   [StartValue {LazyNumberGenerator StartValue + 1}]
end
```

b) The `{LazyNumberGenerator StartValue}` function implements a lazy infinite sequence of integers. It takes a starting value `StartValue` as an argument and returns a list with two elements: The head, which is `StartValue` (the current value), and the tail, which is a lazy, unevaluated call to generate the next value in the sequence. This lazy call increments the current value by `1` for the subsequent iteration.

The solution has several limitations worth noting. Firstly, while it is lazy, accessing many elements could create a long chain of unevaluated functions, potentially consuming significant memory. Depending on the Oz environment and system resources, deep recursion may lead to stack overflow. Additionally, there is no way to terminate the sequence once the function is executed, which could pose issues in certain scenarios.

Task 6

a)

```
fun {Sum List}
   fun {DoSum List Acc}
      case List
      of nil then Acc
      [] Head|Tail then {DoSum Tail Acc}
      end
   end
in
   {DoSum List 0}
end
```

To make the `Sum` function tail recursive, we can introduce a nested helper function `DoSum` that takes an additional accumulator `Acc` parameter to store the running sum of the `List`. We can then modify the base case to return the accumulator instead of `0`, and change the recursive call to be the last operation in the function, passing the updated accumulator. The main `Sum` function now calls `DoSum` with the initial list and an accumulator value of `0`.

b) Tail recursion in Oz offers significant memory efficiency, as the compiler can optimize tail recursive functions to use constant stack space, regardless of input size. This optimization replaces function calls with jump instructions, improving performance by eliminating the overhead of stack frame allocations. Furthermore, it encourages a functional programming style, promoting immutability and a more declarative approach to writing functions.

c) Not all programming languages that support recursion benefit equally from tail recursion. Languages like Clojure, Elixir, Erlang, and Haskell, which have built-in tail call optimization[1], gain significant advantages from tail-recursive functions. In contrast, languages such as C, Go, and Python, which lack guaranteed tail call optimization, do not experience the same benefits.

[1] `https://en.wikipedia.org/wiki/Tail_call#Language_support`