Programmeringsspråk TDT4165 - Assignment 4

Ola Kristoffer Hoff

8. November 2022

Threads

Task 1

In the sequential model of computation, statements are executed sequentially, in a single computation. The thread ... end statement extends the model with concurrency.

a Execute the following code in Mozart and observe the results. What sequence of numbers gets printed as output of the Oz environment?

```
local A=10 B=20 C=30 in
   {System.show C}

   thread
      {System.show A}
      {Delay 100}
      {System.show A * 10}
   end

   thread
      {System.show B}
      {Delay 100}
      {System.show B * 10}
   end

   {System.show C * 100}
end
```

Figure 1: The code to be run in task 1 a.

```
30
10
20
3000
200
100
```

Figure 2: One potential output from task 1 a.

b Explain with your own words how execution proceeds and why the result is as such. Would it be possible to have a different sequence printed as output? Explain your answer.

As we can see in figure 2 we print the C-value first, as expected. Then we start a thread that prints A and then it delays. In the meantime the main program continues and starts the next thread that prints B and delays. Then the main program continues and prints C * 100 before the threads are done delaying. Then the thread for B gets runtime again before thread A and print s B * 10 and lastly thread A prints A * 10.

This is one of many potential sequences of printing. Since we are running threads we have a non-deterministic execution order between the different threads and the main program.

c Execute the following code in Mozart and observe the results. What sequence of numbers gets printed as output of the Oz environment?

```
26 local A B C in
27    thread
28       A = 2
29       {System.show A}
30    end
31    thread
32       B = A * 10
33       {System.show B}
34    end
35
36    C = A + B
37    {System.show C}
38
39 end
```

Figure 3: Code to run in task 1 c.

```
2
20
22
```

Figure 4: Output of running code in figure 3.

d Explain with your own words how execution proceeds and why the result is as such. Would it be possible to have a different sequence printed as output? Explain your answer.

In this case there a deterministic sequence of when the results of A, B and C are ready, but they can be printed in any order.

First we start a threads where A is defined, then we print A. Then a new thread for B is started and calculated B based on A, so this thread will stall on this line (32) until A is resolved (at line 28).

The same case goes for C (line 36), it depends on A and B. So the order in which A, B and C is resolved is A, B and C. However, it is possible that the threads are suspended before they can print, but after they have resolved their variable. In this case the printing could happen in another order.

Streams

A stream is a list that is created incrementally by leaving the tail as an unbound dataflow variable: it is extended by binding that variable to the next value, and then appending a new unbound tail. Streams are potentially infinite, and have various applications in the processing of sequence of data of unspecified length. By combining streams and threads it is possible to implement a producer/consumer model in a straightforward way, thanks to declarative concurrency.

Task 2

a Implement a function fun {Enumerate Start End} that generates, asynchronously, a stream of numbers from Start until End.

{System.show {Enumerate 1 5}} should print [1 2 3 4 5]

Hint: You can use the thread ... end statement inside the definition of the function, to wrap the iterative process that generates the numbers.

```
declare Enumerate

fun {Enumerate Start End}

       if Start > End then
          nil
       else
          Start | thread {Enumerate Start + 1 End} end
       end
end
```

Figure 5: The code for asynchronously generating a list from a start to end value.

In figure 5 we can see the code. To print the list with "{System.show {Enumerate 1 5 }}" and obtaining "[1 2 3 4 5]" as the output did not happen. I instead got "1 — _<optimized>". I could use "{Browse {Enumerate 1 5 }}" to get the correct result, but after a visit to piazza I saw the instructor had written what is shown in figure **??** and so I ended up printing it with "{System.show {List.take {Enumerate 1 5} 5}}".

the instructors' answer, *where instructors collectively construct a single answer*

Yes, the first version would be fine.

The answer to your question about `_<optimized>` should be in the Appendix of the assignment PDF. You need to explicitly "query" the values after the first to visualize them. You can try `{List.take}` with the number of values you want, or otherwise the function in the Appendix.

For example, the following will return a "normal" list with the first 5 values taken from the "MyStream" list:
`{List.take MyStream 5}`

thanks! | 0 Updated 1 week ago by Leonardo Montecchi

Figure 6: Screenshot from piazza of answer to question about printing in task 2 a.

b Implement a function fun {GenerateOdd Start End} that generates, asynchronously, a stream of odd numbers from Start to End. The GenerateOdd function must be implemented as a consumer of Enumerate. That is, it must read the stream generated by Enumerate and filter it as appropriate.

{System.show {GenerateOdd 1 5}} should print [1 3 5]

{System.show {GenerateOdd 4 4}} should print nil

```
declare GenerateOdd ConsumeList Filter

fun {Filter X}
   if X mod 2 \= 0 then
      true
   else
      false
   end
end

fun {ConsumeList List}
   case List of nil then
      nil
   [] Head|Tail then
      if {Filter Head} == true then
         Head | thread {ConsumeList Tail} end
      else
         thread {ConsumeList Tail} end
      end
   end
end

fun {GenerateOdd Start End}
   {ConsumeList thread {Enumerate Start End} end}
end
```

Figure 7: Code for generating a list of odd numbers in a given inclusive range asynchronously.

The same case as described in the previous task with the printing happened here as well. So again I printed it using the "List.take"-method.

Task 3

In this task we will implement a generator of prime numbers, exploiting Oz streams and concurrency.

a Implement the function fun {ListDivisorsOf Number}, which produces a stream of all the divisors of the integer number Number. A number $d \in N$ is a divisor of $n \in N$ if the rest of the integer division $\frac{n}{d}$ is zero. The modulo operation (i.e., rest of integer division) is denoted with the keyword mod in Oz. ListDivisorOf must be implemented as a consumer of Enumerate.

```
declare ListDivisorsOf ListDivisorsOfConsumer IsDivisor

fun {IsDivisor Number Divisor}
   if Number mod Divisor == 0 then
      true
   else
      false
   end
end

fun {ListDivisorsOfConsumer List Number}
   case List of nil then
      nil
   [] Head|Tail then
      if {IsDivisor Number Head} then
         Head | thread {ListDivisorsOfConsumer Tail Number} end
      else
         thread {ListDivisorsOfConsumer Tail Number} end
      end
   end
end

fun {ListDivisorsOf Number}
   {ListDivisorsOfConsumer {Enumerate 1 Number} Number}
end
```

Figure 8: Code for *"ListDivisorsOf"*-function.

b Implement the function fun {ListPrimesUntil N}, which produces a stream of all the prime numbers up to the number N. A number n is prime if its only divisors are 1 and n itself. ListDivisorOf must be implemented as a consumer of Enumerate.

Hint: You can chain multiple streams, and also consume multiple streams in the implementation of a function. In particular, you should also consume the stream produced by ListDivisorsOf.

```
declare ListPrimesUntil ConsumePrimes

fun {ConsumePrimes List}
   case List of Head|Tail then
      case {ListDivisorsOf Head} of H|T then
         if {And H == 1 T == nil} then
            Head | thread {ConsumePrimes Tail} end
         else if {And H == 1 T.1 == Head} then
            Head | thread {ConsumePrimes Tail} end
         else
            thread {ConsumePrimes Tail} end
              end
         end
      end
   [] nil then
      nil
   end
end

fun {ListPrimesUntil N}
   {ConsumePrimes thread {Enumerate 1 N} end}
end
```

Figure 9: Code for finding all primes up until N.

Lazy Evaluation

Task 4

The lazy keyword can be applied to functions to specify that they will be evaluated lazily, meaning that the values would be computed only when needed. This is particularly useful for working with (potentially) infinite streams.

We can rewrite our generator of prime numbers as a lazy function, using the lazy annotation.

a Implement a function fun {Enumerate} as a lazy function that generates an infinite stream of numbers, starting from 1.

```
declare EnumerateLazy EnumerateLazyConditioned

fun lazy {EnumerateLazy}

   fun lazy {EnumerateLazyConditioned N}
      N | {EnumerateLazyConditioned N + 1}
   end

   1 | {EnumerateLazyConditioned 2}
end
```

Figure 10: Code for lazy function. to generate an infinite stream of numbers starting at 1.

b Implement a function fun {Primes} as a lazy function that generates an infinite stream of prime numbers, starting from 2. You must implement Primes as a consumer of the stream produced by Enumerate, and any other streams you find useful.

Note: Streams produced by lazy functions are not easily displayed using the usual Browse procedure, because browsing a stream does not count as "needing" its value. Similarly, using the System.show procedure will just show <optimized>. To actually visualize the content of the stream you need to access its values, e.g., {List.take {Enumerate} 10} will show the first 10 values of the stream produced by Enumerate. See also the Appendix for additional hints on how to work with streams.

```
declare Primes ConsumePrimesLazy

fun {ConsumePrimesLazy List}
   case List of Head|Tail then
      if Head == 1 then
         {ConsumePrimesLazy Tail}
      else
         case {ListDivisorsOf Head} of H|T then
            if {And H == 1 T == nil} then
               Head | {ConsumePrimesLazy Tail}
            elseif {And H == 1 T.1 == Head} then
               Head | {ConsumePrimesLazy Tail}
            else
               {ConsumePrimesLazy Tail}
            end
         [] nil then
            nil
         end
      end
   [] nil then
      nil
   end
end

fun lazy {Primes}
   {ConsumePrimesLazy {EnumerateLazy}}
end
```

Figure 11: Code for generating primes with a lazy number generator. Functions used that are not defined in the figure is from earlier figures.