TDT4165 - Programming Languages

Assignment 4: Declarative Concurrency

Benjamin Zubača

October 30, 2024

Task 1

a) In the code

```
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
```

the following output is produced:

```
30      % {System.show C}
3000    % {System.show C * 100}
10      % {System.show A}
20      % {System.show B}
200     % {System.show B * 10}
100     % {System.show A * 10}
```

b) To understand this order, we can examine how the program executes according to the declarative concurrent model of computation: After pushing both `thread end` blocks onto their own semantic stacks, the scheduler decides the next stack to execute, introducing some non-determinism. Thus, the order of output depends on the scheduler's choices, which may vary. This is why alternative sequences could potentially appear in the output, depending on the thread execution order and delays.

c) The code

```
local A B C in
   thread
      A = 2
      {System.show A}
   end

   thread
```

```
8         B = A * 10
9         {System.show B}
10     end
11
12     C = A + B
13     {System.show C}
14  end
```

produces the following order:

```
2   % {System.show A}
20  % {System.show B}
22  % {System.show C}
```

d) This ordering results from Oz's use of data flow variables: `System.show` calls involving these variables will cause their respective threads to pause if the variables are not yet bound. Threads only proceed once the necessary variables are fully resolved.

In this code, the first thread assigns `A = 2` and displays it, which allows the second thread to compute `B` as `A * 10` and display the result. Finally, `C` is calculated as `A + B`, so `{System.show C}` displays the result once both `A` and `B` are bound. This data flow synchronization ensures a consistent order of output, meaning that a different sequence would not be possible.

Task 2

a)

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

The `{Enumerate Start End}` function generates a stream of numbers from `Start` to `End`. Each recursive call runs in a new thread, creating the list one element at a time. The current number (`Start`) is returned immediately while the rest of the list (`Start+1` to `End`) is computed in parallel. The function terminates when `Start` exceeds `End`.

b)

```
fun {GenerateOdd Start End}
   fun {IsOdd Number}
      Number mod 2 == 1
   end
in
   thread {Filter {Enumerate Start End} IsOdd} end
end
```

The `{GenerateOdd Start End}` function creates a stream of odd numbers between `Start` and `End`. It leverages the `Enumerate` stream by filtering it through the `IsOdd` predicate, which checks if a number is odd using `mod 2`. The threading ensures asynchronous generation, where each odd number is produced on demand while the rest of the sequence is computed in parallel.

c) Running `{Show Enumerate 1 5}` and `{Show GenerateOdd 1 5}` sequentially, we get the following output:

```
[1 2 3 4 5]
_<optimized>
```

When using `Show`, we observe different outputs for our functions. `{Show {Enumerate 1 5}}` displays `[1 2 3 4 5]` probably because the main thread waits for all elements to be generated before showing the result. In contrast, `{Show {GenerateOdd 1 5}}` displays `_<optimized>` because it creates a seperate thread for filtering, and this thread computation is not yet complete when `Show` is called. Oz recognizes this as a pending computation and displays `_<optimized>` instead of waiting.

Task 3

a)

```
fun {ListDivisorsOf Number}
   fun {IsDivisorOf X Y}
      Y mod X == 0
   end
in
   thread
      {Filter {Enumerate 1 Number}
       fun {$ X} {IsDivisorOf X Number} end}
   end
end
```

The function `{ListDivisorsOf Number}` consumes the `Enumerate` stream to find all divisors of a given number. It uses a filter with a predicate that checks if each potential divisor divides the input number evenly, i.e, with no remainder using the `mod` operator. The function stops checking numbers larger than the input.

b)

```
fun {IsPrime Number}
   case {ListDivisorsOf Number}
   of [1 Number] then true
   else false
   end
end

fun {ListPrimesUntil Number}
   thread {Filter {Enumerate 2 Number} IsPrime} end
end
```

The `{ListPrimesUntil Number}` also consumes the `Enumerate` stream, but filters for prime numbers up to `Number`. It determines if a number is prime by using `ListDivisorsOf` to count its divisors; a number is prime if and only if it has exactly two divisors (1 and itself). The function stops once it reaches `Number`.

Task 4

a)

```
fun lazy {Enumerate}
   fun lazy {EnumerateFrom Number}
      Number|{EnumerateFrom Number + 1}
   end
in
   {EnumerateFrom 1}
end
```

The `{Enumerate}` lazy function generates an infinite stream of natural numbers starting from 1. It achieves this through a helper lazy function `{EnumerateFrom Number}` that creates a stream where `Number` is the head, followed by recursively generating numbers from `Number + 1` onwards. The main function initiates this enumeration from 1.

b)

```
fun lazy {Primes}
   fun lazy {LazyFilter List Pred}
      case List
      of Head|Tail then
         if {Pred Head} then Head|{LazyFilter Tail Pred}
         else {LazyFilter Tail Pred} end
      end
   end
in
   {LazyFilter {Enumerate} IsPrime}
end
```

The `Primes` function generates an infinite stream of prime numbers using lazy evaluation. It uses a `{LazyFilter}` helper lazy function that processes a list `List` according to the predicate function `Pred`. Rather than computing all values at once, it evaluates each element only when needed, retaining numbers that satisfy the predicate function `IsPrime`.