TDT4165 PROGRAMMING LANGUAGES

Exercise 4

Declarative Concurrency

Fall 2019

Preliminaries

This exercise is about streams, threads, lazy programming and declarative concurrency.

Relevant reading: Chapter 4.1-4.5 in CTMCP.

To be able to compile your code, it must be in the form:

```
functor import Application System define
    % Your code here
    {Application.exit 0}
end
```

Compile with

```
ozc -c your-filename.oz
ozengine your-filename.ozf
```

If you do not call {`Application.exit 0`} (or your program hangs) then you can kill the it using the ctrl+c key combination.

You can deliver one file with all the code and theory answers as comments in that file.

Evaluation

You will get a score from 0 to 100 on this exercise. This score will later be converted into a final assignment score. Information on how this conversion is done can be found on Blackboard. Points for each task and sub-task is stated in the exercise. For the programming tasks you will be awarded full score for a correct implementation. Points may be deducted from this score as a percentage of the available points for the given task:

- Code does not run without modifications, but is otherwise correct. (20 % deduction)
- Modules other than the System, OS and Application modules are imported and used. (100 % deduction)
- The code does not produce the output requested by the task and example output, but is mostly correct. (50 % deduction)
- The implementation is correct but is overly complex, long or redundant. (20 % deduction)
- Unreasonable indentation of code. (20 % deduction)
- Functions, procedures and identifiers have names that are not meaningful. (20 % deduction)

Streams and Threads

Task 1 (10 p)

A stream is a list that is created incrementally by leaving the tail as an unbound dataflow variable. Implement `fun {GenerateOdd S E}` which generates a stream of odd numbers between S and E in ascending order.
`{System.show {GenerateOdd ~3 10}}` should print `[~3 ~1 1 3 5 7 9]`.
`{System.show {GenerateOdd 3 3}}` should print `[3]`.
`{System.show {GenerateOdd 2 2}}` should print `nil`.

Task 2 (10 p)

Implement `fun {Product S}` which returns the product of all elements in the list S.
`{System.show {Product [1 2 3 4]}}` should print `24`.

Task 3 (15 p)

Generate a stream of odd numbers between 0 and 1000 on one thread, and multiply the elements of the stream (fold/reduce the stream) on a separate thread. What are the first three digits of the output? What is the benefit of running on two separate threads?

Lazy Execution

Task 4 (15 p)

Rewrite your function from task 1 to be lazy, using the `lazy` annotation. How does this affect task 3 in terms of throughput and resource usage?

Bounded Buffer

Task 5 (50 p)

You have just started a new hammer business. You have a hammer factory that produces hammers, and a consumer that buys the hammers. To make sure you don't produce more hammers than the consumer needs, you have decided to go for just-in-time delivery. Only producing hammers when the consumer orders them.

a) Create a lazy hammer factory `fun lazy {HammerFactory}` that returns a stream of hammers. A hammer is either the atom 'working' or the atom 'defect'. The hammer should be 'working' in 90 % of the cases, and 'defect' in 10 % of the cases. The factory produces 1 hammer each second.
A list with 4 working and 1 defect hammers could look like [working working working working defect] (15 p)

To generate a random number, you can import OS and use this function:

```
fun {RandomInt Min Max}
    X = {OS.rand}
    MinOS
    MaxOS
in
    {OS.randLimits ?MinOS ?MaxOS}
    Min + X*(Max - Min) div (MaxOS - MinOS)
end
```

Try your solution using:

```
local HammerTime B in
    HammerTime = {HammerFactory}
    B = HammerTime.2.2.2.1
    {System.show HammerTime}
end
```

This should after a 4 second delay print something like:
`working|defect|working|working|_<optimized>`.

b) The consumer is ordering N hammers from the factory, and counts the amount of working hammers. Implement the function {HammerConsumer HammerStream N} that takes a stream and a number, reads the N first elements from the HammerStream, and returns the amount of working hammers. (15 p)

Test your solution using

```
local HammerTime Consumer in
    HammerTime = {HammerFactory}
    Consumer = {HammerConsumer HammerTime 10}
    {System.show Consumer}
end
```

This should after a 10 second delay print a number. On average the number should be 9.

c) You have a gotten a lot of complaints. Ordering hammers takes way too long. With the lazy HammerFactory, the customer has to wait for the factory to make the hammer after ordering it. To solve this problem you have decided to make a bounded buffer for storing some hammers ahead of time. Implement `fun {BoundedBuffer HammerStream N}` that takes a stream of hammers and returns a bounded buffer with size N Explain what is happening in the bounded buffer with comments in the code. (20 p)

Try your solution with and without the buffer to see the difference in execution time.
With buffer:

```
local HammerStream Buffer in
    HammerStream = {HammerFactory}
    Buffer = {BoundedBuffer HammerStream 6}
    {Delay 6000}
    Consumer = {HammerConsumer Buffer 10}
    {System.show Consumer}
end
```

Without buffer:

```
local HammerStream in
    HammerStream = {HammerFactory}
    {Delay 6000}
    Consumer = {HammerConsumer HammerStream 10}
    {System.show Consumer}
end
```

The execution time should be approximately 10 and 16 seconds.