Homework 4: Declarative Concurrency

Due: problems 1, 3–5, Monday, October 29, 2007 at 11pm; problems 7–21, Monday, November 5, 2007 at 11pm

In this homework you will learn about the declarative concurrent model and basic techniques of declarative concurrent programming. The programming techniques include stream programming and lazy functional programming [Concepts] [UseModels]. A few problems also make comparisons with the declarative model and with concurrency features in Java [EvaluateModels] [MapToLanguages].

Don't use side effects (assignment and cells) in your Oz solutions (however, you can use them in the Java code you write).

You should use helping functions whenever you find that useful. Unless we specifically say how you are to solve a problem, feel free to use any functions from the Oz library (base environment), especially functions like `Map` and `FoldR`.

For all programing tasks, you must run your code using the Mozart/Oz system. For these you must also provide evidence that your program is correct (for example, test cases). Oz code with tests for various problems is available in a zip file, which you can download from the course resources web page. For testing, you may want to use tests based on our code in the course library file `TestingNoStop.oz`.

Turn in (on WebCT) your code and, if necessary, output of your testing for all questions that require code. Please upload text files with your code that have the suffix `.oz` (or `.java` as appropriate), and text files with suffix `.txt` that contain the output of your testing. Please use the name of the function as the name of the file. (In any case, don't put any spaces in your file names!)

If we provide tests and your code passes all of them, you can just indicate with a comment in the code that your code passes all the tests. Otherwise it is necessary for you to provide us test code and/or test output. If the code does not pass some tests, indicate in your code with a comment what tests did not pass, and try to say why, as this write enhances communication and makes commenting on the code easier and more specific to your problem than just leaving the buggy code without comments.

If you're not sure how to use our testing code, ask us for help.

Your code should compile with Oz, if it doesn't you probably should keep working on it. If you don't have time, at least tell us that you didn't get it to compile.

Be sure to clearly label what problem each area of code solves with a comment.

You might loose points even if your code is passing all the tests, if you don't follow the grammar or if your code is confusing or needlessly inefficient. Make sure you don't have extra case clauses or unnecessary if tests, make sure you are using as much function definitions/calls as needed, check if you are making unnecessary loops or passes in your code, efficient code means full mark, working code does not necessarily mean full mark.

If you are using Windows Vista, then you cannot use `Browse` and must instead use `Show` (or to see strings, `System.showInfo`). However, this is somewhat problematic because `Show` (and `showInfo`) only show you the instantaneous view of their argument. In essence there will be a race condition between other threads and the thread using `Show`. One way around this is to use the `Wait` operation in Oz (see Section 10.1 of "The Oz Base Environment" [DKS06]). For example, write `{Wait X} {Show X}` where the textbook has `{Browse X}`. In cases where you don't have a single variable to wait on, you can introduce delays using the `Delay` procedure (see Section 10.3 of "The Oz Base Environment" [DKS06]). For example, use `{Delay 5000} {Show Lst}` where code in the textbook shows `{Browse Lst}`. You may have to adjust the timing argument to delay (which is measured in milliseconds) to fit the problem.

Don't hesitate to contact the staff if you are stuck at some point.

Read Chapter 4 of the textbook [RH04]. (See the syllabus for optional readings.)

Thread and Dataflow Behavior Semantics

The following problems explore the semantics of the declarative concurrent model.

1. (20 points) [Concepts]

   Do problem 1 in section 4.11 of the textbook [RH04] (Thread Semantics). Note that part (b) should read (according to the errata) "All of these executions . . .".

2. (10 points; extra credit) [Concepts]

   Do problem 2 in section 4.11 of the textbook [RH04] (Threads and garbage collection).

3. (10 points) [Concepts]

   Do problem 4 in section 4.11 of the textbook [RH04] (Order-determining concurrency).

4. This problem explores threads and dataflow in Java [UseLanguages] [MapToLanguages]

   For more informtion about threads in Java see, for example, *The Java Tutorial* [CW98], which is online at `http://java.sun.com/docs/books/tutorial/`. The concurrency section of that tutorial has several examples.

   (a) (10 points) Write, in Java, write a version of the program in the textbook's problem 4 (of section 4.11) in which the Oz program's dataflow variables `A`, `B`, `C`, and `D` are represented by fields (perhaps static fields) in the Java code. Call your Java class `FourThreads`.

   Hints: You will need to either use package-visible fields or nest your subclasses of `Thread` or subtypes of `Runnable` within your main `FourThreads` class. The use of anonymous inner classes will make your code shorter.

   (b) (5 points) How much extra code is present in the Java version compared with the Oz code?

   (c) (5 points) Does your Java code have the possibility of race conditions? Briefly explain.

   (d) (5 points) Does your Java code always give the same answer as the Oz Code? Briefly explain.

   (e) (40 points) Write, in Java, a class `IntDataflowVariable`, with operations `int get()` and `void set(int)`. The `get` operation should suspend the calling thread until the `set` method is called. The `set` method should throw an `IllegalArgumentException` if it is called again with a different argument than the value used the first time it was called. Test by using it to write another version of the textbook's problem 4 (of section 4.11) using instances of your class `IntDataflowVariable` in place of the integer fields. Call this class `FourThreadsWithDataflow`. Running this class's main method should always give the same output as the Oz code.

5. (10 points) [Concepts] [UseModels]

   Do problem 5 in section 4.11 of the textbook [RH04] (The Wait Operation).

6. (20 points; extra credit) [Concepts]

   Do problem 7 in section 4.11 of the textbook [RH04] (Programmed triggers using higher-order programming).

7. (20 points) [Concepts]

   Do problem 8 in section 4.11 of the textbook [RH04] (Dataflow behavior in a concurrent setting).

Streams and Lazy Functional Programming

The following problems, many of which are due to John Hughes, relate to modularization of numeric code using streams and lazy execution. In particular, we will explore the Newton-Raphson algorithm. This algorithm computes better and better approximations to the square root of a floating point number `N` from a previous approximation `X` by using the following curried function.

```
% $Id: Next.oz,v 1.2 2007/10/31 00:42:36 leavens Exp leavens $
declare
% Next: <fun {$ <Float>}: <fun {$ <Float>}: Float>>
fun {Next N}
   fun {$ X}
      (X + N / X) / 2.0
   end
end
```

In the following we will use the type `<Stream T>` as a synonym for the type `<List T>`, in which the list is presumed to be infinite. Note that `nil` never occurs in a `<Stream T>`. Thus the grammar for `<Stream T>` is as follows.

⟨Stream T⟩ ::= `T` `'|'` ⟨Stream T⟩

As an aid to writing code for this section, and for testing that code, we provide a library file containing predicates for approximate comparisons of floating point numbers and for testing with approximate comparisons. The floating point approximate comparison code is shown in Figure 1 on the following page. The testing code for floating point numbers is shown in Figure 2 on page 5.

8. (10 points) [Concepts] [UseModels]

   Write a lazy function

   `StreamIterate: <`**`fun`**` lazy {$ <`**`fun`**` {$ T}: T> T}: <Stream T>>`

   such that `{StreamIterate` F X`}` takes a function F and a value X and returns the stream

   $$X \mid \{FX\} \mid \{F\{FX\}\} \mid \{F\{F\{FX\}\}\} \mid \ldots,$$

   that is, the stream whose i^{th} item, counting from 1, is F^{i-1} applied to X.

   The examples in Figure 3 on page 6 are written using the `WithinTest` procedure from Figure 2 on page 5. Notice also that, since `Next` is curried, we don't pass `Next` itself to `StreamIterate`, but instead pass the value of applying `Next` to some number.

9. (5 points) [Concepts]

   Why does `StreamIterate` (see Exercise 8) have to be lazy? Give a brief answer.

10. (10 points) [UseModels]

    Write a function

    `Approximations: <`**`fun`**` {$ Float Float}: <Stream Float>>`

    such that `{Approximations N A0}` returns the infinite list of approximations to the square root of `N`, starting with `A0`, according to the Newton-Raphson method, that is by iterating `{Next N}`.

11. (5 points) [Concepts] [UseModels]

    Does your solution to Exercise 10 have to be lazy? Briefly explain.

12. (10 points) [UseModels]

    Write a function

    `ConvergesTo: <`**`fun`**` {$ <Stream T> <`**`fun`**` {$ T T}: Bool>}: T>`

    such that `{ConvergesTo Xs Pred}` looks down the stream `Xs` to find the first two consecutive elements of `Xs` that satisfy `Pred`, and it returns the second of these consecutive elements. (It will never return if there is no such pair of consecutive elements.) Figure 5 on page 6 gives some examples.

```
% $Id: FloatPredicates.oz,v 1.3 2007/10/23 02:14:21 leavens Exp leavens $
%% Some functions to do approximate equality of floating point numbers.
%% AUTHOR: Gary T. Leavens

declare
%% Return true iff the difference between X and Y
%% is no larger than Epsilon
fun {Within Epsilon X Y} {Abs X-Y} =< Epsilon end

%% Partly curried version of Within
fun {WithinMaker Epsilon} fun {$ X Y} {Within Epsilon X Y} end end

%% Return true iff the corresponding lists are
%% equal relative to the given predicate
fun {CompareLists Pred Xs Ys}
   case Xs#Ys of
      nil#nil then true
   [] (X|Xr)#(Y|Yr) then {Pred X Y} andthen {CompareLists Pred Xr Yr}
   else false
   end
end

%% Return true iff the lists are equal
%% in the sense that the corresponging elements
%% are equal to within Epsilon
fun {WithinLists Epsilon Xs Ys}
   {CompareLists {WithinMaker Epsilon} Xs Ys}
end

%% Return true iff the ratio of X-Y to Y is within Epsilon
fun {Relative Epsilon X Y} {Abs X-Y} =< Epsilon*{Abs Y} end

%% Partly curried version of Relative
fun {RelativeMaker Epsilon} fun {$ X Y} {Relative Epsilon X Y} end end

%% Return true iff the lists are equal
%% in the sense that the corresponging elements
%% are relatively equal to within Epsilon
fun {RelativeLists Epsilon Xs Ys}
   {CompareLists {RelativeMaker Epsilon} Xs Ys}
end

%% A useful tolerance for testing
StandardTolerance = 1.0e~3

%% A convenience for testing, relative equality with a fixed Epsilon
ApproxEqual = {RelativeMaker StandardTolerance}
```

Figure 1: Comparisons for floating point numbers. This code is available in the course `lib` directory.

```
% $Id: FloatTesting.oz,v 1.4 2006/10/27 08:57:49 leavens Exp $
% Testing for floating point numbers.
% AUTHOR: Gary T. Leavens

\insert 'FloatPredicates.oz'
\insert 'TestingNoStop.oz'

declare
%% TestMaker returns a procedure P such that {P Actual '=' Expected}
%% is true if {FloatCompare Epsilon Actual Expected} (for Floats)
%% or if {FloatListCompare Epsilon Actual Expected} (for lists of Floats)
%% If so, print a message, otherwise throw an exception.
fun {TestMaker FloatCompare FloatListCompare Epsilon}
   fun {Compare Actual Expected}
      if {IsFloat Actual} andthen {IsFloat Expected}
      then {FloatCompare Epsilon Actual Expected}
      elseif {IsList Actual} andthen {IsList Expected}
      then {FloatListCompare Epsilon Actual Expected}
      else false
      end
   end
in
   proc {$ Actual Connective Expected}
      if {Compare Actual Expected}
      then {System.showInfo
            {Value.toVirtualString Actual 5 20}
            # ' ' # Connective # ' '
            # {Value.toVirtualString Expected 5 20}}
      else {System.showInfo
            'TEST FAILURE: '
            # {Value.toVirtualString Actual 5 20}
            # ' ' # Connective # ' '
            # {Value.toVirtualString Expected 5 20}
           }
      end
   end
end

WithinTest = {TestMaker Within WithinLists StandardTolerance}
RelativeTest = {TestMaker Relative RelativeLists StandardTolerance}
```

Figure 2: Testing code for floating point. This puts output on standard output (the `*Oz Emulator*` window). The file `FloatPredicates` is shown in Figure 1 on the previous page. This file is available in the course `lib` directory. To use it, copy the files from the course directory to your own directory and then put `\insert 'FloatTesting.oz'` in your file.

```
% $Id: StreamIterateTest.oz,v 1.6 2007/10/31 00:40:15 leavens Exp leavens $
\insert 'StreamIterate.oz'
\insert 'Next.oz'
\insert 'FloatTesting.oz'
{StartTesting 'StreamIterate...'}
{WithinTest {List.take {StreamIterate {Next 1.0} 1.0} 7}
 '~=~' [1.0 1.0 1.0 1.0 1.0 1.0 1.0]}
{WithinTest {List.take {StreamIterate {Next 9.0} 1.0} 7}
 '~=~' [1.0 5.0 3.4 3.0235 3.0001 3.0 3.0]}
{WithinTest {List.take {StreamIterate {Next 200.0} 1.0} 7}
 '~=~' [1.0 100.5 51.245 27.574 17.414 14.449 14.145]}
{RelativeTest {List.take {StreamIterate {Next 0.144} 7.0} 9}
 '~=~' [7.0 3.5103 1.7757 0.92838 0.54174 0.40378 0.3802 0.37947 0.37947]}
{RelativeTest {List.take {StreamIterate fun {$ X} X*X end 2.0} 9}
 '~=~' [2.0 4.0 16.0 256.0 65536.0 4.295e009 1.8447e019 3.4028e038
        1.1579e077]}
{RelativeTest {List.take {StreamIterate fun {$ X} X/3.0 end 10.0} 8}
 '~=~' [10.0 3.3333 1.1111 0.37037 0.12346 0.041152 0.013717 0.0045725]}
```

Figure 3: Tests for Exercise 8 on page 3.

```
% $Id: ApproximationsTest.oz,v 1.2 2007/10/31 00:46:25 leavens Exp leavens $
\insert 'Approximations.oz'
\insert 'FloatTesting.oz'
{StartTesting 'Approximations...'}
{WithinTest {List.take {Approximations 1.0 1.0} 7}
 '~=~' [1.0 1.0 1.0 1.0 1.0 1.0 1.0]}
{RelativeTest {List.take {Approximations 2.0 1.0} 5}
 '~=~' [1.0 1.5 1.41667 1.41422 1.41421]}
{RelativeTest {List.take {Approximations 64.0 1.0} 5}
 '~=~' [1.0 32.5 17.2346 10.474 8.29219]}
```

Figure 4: Tests for Exercise 10 on page 3.

```
% $Id: ConvergesToTest.oz,v 1.5 2007/10/31 00:45:52 leavens Exp leavens $
\insert 'ConvergesTo.oz'
\insert 'FloatTesting.oz'

fun lazy {Repeat X} X|{Repeat X} end

{StartTesting 'ConvergesTo...'}
{WithinTest {ConvergesTo
             {Append [1.0 3.5 4.5] {Repeat 7.0}}
             {WithinMaker 1.01}
            }
 '~=~' 4.5}
{WithinTest {ConvergesTo
             {Append [1.0 32.5 17.2346 10.474 8.29219 8.00515] {Repeat 8.0}}
             {WithinMaker 0.5}
            }
 '~=~' 8.00515}
```

Figure 5: Tests for Exercise 12 on page 3.

13. (10 points) [UseModels]

    Using `WithinMaker` from Figure 1 on page 4 and other functions given above, write a function

    ```
    SquareRoot: <fun {$ Float Float Float}: Float>
    ```

    such that `{SquareRoot A0 Epsilon N}` returns an approximation to the square root of `N` that is within `Epsilon`. The parameter `A0` is used as an initial guess in the Newton-Raphson method. Examples are given in Figure 6 on the next page.

14. (10 points) [UseModels]

    Using `RelativeMaker` from Figure 1 on page 4 and the other pieces given above, write a function

    ```
    RelativeSquareRoot: <fun {$ Float Float Float}: Float>
    ```

    such that `{RelativeSquareRoot A0 Epsilon N}` returns an approximation to the square root of `N` that only has a relative error of `Epsilon`. The parameter `A0` is used as an initial guess in the Newton-Raphson method. When iterating, keep going until the ratio of the difference between the last and the previous approximation to the last approximation approaches 0, instead of waiting for the differences between the approximations themselves to approach zero. (This is equivalent to iterating until the ratio of the last two approximations approaches 1.) This is test for convergence is better for square roots of very large numbers, and for square roots of very small numbers. Examples are given in Figure 7 on the following page.

15. (15 points) [UseModels]

    You may recall that the derivative of a function `F` at a point `X` can be approximated by the following function.

    ```
    % $Id: EasyDiff.oz,v 1.3 2007/10/31 00:44:27 leavens Exp leavens $
    declare
    fun {EasyDiff F X Delta} ({F (X+Delta)} - {F X}) / Delta end
    ```

    Good approximations are given by small values of `Delta`, but if `Delta` is too small, then rounding errors may swamp the result. One way to choose `Delta` is to compute a sequence of approximations, starting with a reasonably large one. If `{WithinMaker Epsilon}` is used to select the first approximation that is accurate enough, this can reduce the risk of a rounding error affecting the result.

    Your task is to write a function

    ```
    DiffApproxims: <fun {$ Float <fun {$ Float}: Float> Float}:
                         <Stream Float>>
    ```

    such that `{DiffApproxims Delta0 F X}` returns an infinite list of approximations to the derivative of `F` at `X`, where at each step, the current `Delta` is halved. Examples are given in Figure 8 on the following page.

    Hint: do you need to make something lazy to make this work?

16. (15 points) [UseModels]

    Write a function

    ```
    Differentiate: <fun {$ Float Float  <fun {$ Float}: Float> Float}:
                         Float>
    ```

    such that `{Differentiate Epsilon Delta0 F N}` returns an approximation that is accurate to within `Epsilon` to the derivative of `F` at `N`. Use the previous problem (Exercise 15) with `Delta0` as the initial value for `Delta`. Examples are given in Figure 9 on page 9.

```
% $Id: SquareRootTest.oz,v 1.2 2007/10/31 00:35:49 leavens Exp leavens $
\insert 'SquareRoot.oz'
\insert 'FloatTesting.oz'
{StartTesting 'SquareRoot...'}
Millionth = 0.0000001
WithinMillionth = {TestMaker
                   Within
                   WithinLists
                   Millionth}
{WithinMillionth {SquareRoot 1.0 Millionth 2.0} '~=~' 1.41421356}
{WithinMillionth {SquareRoot 1.0 Millionth 4.0} '~=~' 2.0}
{WithinMillionth {SquareRoot 1.0 Millionth 64.0} '~=~' 8.0}
{WithinMillionth {SquareRoot 1.0 Millionth 3.14159} '~=~' 1.7724531}
```

Figure 6: Tests for Exercise 13 on the preceding page.

```
% $Id: RelativeSquareRootTest.oz,v 1.2 2007/10/31 00:36:50 leavens Exp leavens $
\insert 'RelativeSquareRoot.oz'
\insert 'FloatTesting.oz'
{StartTesting 'RelativeSquareRoot...'}
TenThousandth = 0.000001
RelativeTenThousandth = {TestMaker
                   Relative
                   RelativeLists
                   TenThousandth}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 2.0}
                        '~=~' 1.41421356}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 4.0}
                        '~=~' 2.0}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 64.0}
                        '~=~' 8.0}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 3.14159}
                        '~=~' 1.7724531}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 9.0e24}
                        '~=~' 3.0e12}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 9.0e~40}
                        '~=~' 3.0e~20}
```

Figure 7: Tests for Exercise 14 on the previous page.

```
% $Id: DiffApproximsTest.oz,v 1.2 2007/10/31 00:45:11 leavens Exp leavens $
\insert 'DiffApproxims.oz'
\insert 'FloatTesting.oz'
{StartTesting 'DiffApproxims...'}
{WithinTest {List.take {DiffApproxims 500.0 fun {$ X} X*X end 20.0} 9}
 '~=~' [540.0 290.0 165.0 102.5 71.25 55.625 47.8125 43.9062 41.9531]}
{WithinTest {List.take {DiffApproxims 100.0 fun {$ X} X*X*X end 10.0} 8}
 '~=~' [13300.0 4300.0 1675.0 831.25 526.562 403.516 349.316 324.048]}
```

Figure 8: Tests for Exercise 15 on the previous page.

```
% $Id: DifferentiateTest.oz,v 1.2 2007/10/31 00:37:26 leavens Exp leavens $
\insert 'Differentiate.oz'
\insert 'FloatTesting.oz'
{StartTesting 'Differentiate...'}
Millionth = 0.0000001
WithinMillionth = {TestMaker
                   Within
                   WithinLists
                   Millionth}
{WithinMillionth {Differentiate Millionth 500.0 fun {$ X} X*X end 20.0}
 '~=~' 40.0}
{WithinMillionth {Differentiate Millionth 100.0 fun {$ X} X*X*X end 10.0}
 '~=~' 300.0}
```

Figure 9: Tests for Exercise 16 on page 7.

Laziness Problems

The following problems explore more about laziness and its utility.

17. (10 points) [EvaluateModels]

    Do problem 12 in section 4.11 of the textbook [RH04] (Laziness and incrementality).

18. (10 points) [EvaluateModels]

    Do problem 13 in section 4.11 of the textbook [RH04] (Laziness and monolithic functions).

19. (20 points) [Concepts] [UseModels]

    Do problem 16 in section 4.11 of the textbook [RH04] (By-need execution).

20. (15 points) [Concurrency and Exceptions] Do problem 18 in section 4.11 of the textbook [RH04]. (Hint: the "abstract machine semantics" is the operational semantics of Oz, which is described in the textbook's sections 2.4, 2.7.2, 4.1, and 4.9.1. You don't have to do anything formal or especially mathematical with the operational semantics.)

21. (15 points) [Concepts] [EvaluateModels]

    Do problem 19 in section 4.11 of the textbook [RH04] (Limitations of Declarative Concurrency). (Note: there is a typo in the book, the `Merge` function has only two input streams, not three.)

References

[CW98] Mary Campione and Kathy Walrath. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet.* The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[DKS06] Denys Duchier, Leif Kornstaedt, and Christian Schulte. *The Oz Base Environment.* mozart-oz.org, June 2006. Version 1.3.2.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* The MIT Press, Cambridge, Mass., 2004.