Homework 5: Message Passing

Due: exercises 1–2 and 4–6, Monday, November 19, 2007 at 11pm; exercises 7 and 9, Monday, November 26, 2007 at 11pm.

In this homework you will learn about the message passing model and basic techniques of programming in that model. The programming techniques include using port objects [Concepts] [UseModels]. A few problems also make comparisons with the other models we have studied, and also with message passing features in other languages [EvaluateModels] [MapToLanguages].

Your code should be written in the message passing model, so you should not use cells and assignment in your Oz solutions.

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 Oz 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 main 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 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.

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

Read Chapter 5 of the textbook [RH04]. (See the syllabus for optional readings.)

Message Passing Semantics and Expressiveness

1. (30 points) [Concepts] [UseModels]

   Using Oz's message passing model, implement a data abstraction `Box`, by writing the following functions and procedures.

   ```
   NewBox: <fun <$ Value>: Box>
   BoxExchange: <proc <$ Box Value Value>>
   BoxAssign: <proc <$ Box Value>>
   BoxAccess: <fun <$ Box>: Value>
   ```

   A Box is like a Cell, in that it holds a value (of any type). The function call `{NewBox X}` returns a new Box containing the value `X`. The procedure call `{BoxExchange B Old New}` atomically binds `Old` to the value in box `B` and makes `New` be the new value contained in Box `B`. The procedure call `{BoxAssign B V}` makes the value `V` be the new value of Box `B`. The function call `{BoxAccess B}` returns the value contained in Box `B`.

   Your code should pass the tests shown in Figure 1 on the following page

```
% $Id: BoxTest.oz,v 1.1 2007/11/09 21:32:07 leavens Exp leavens $
\insert 'Box.oz'
\insert 'TestingNoStop.oz'
declare B1 B2 V1old V2old
{StartTesting 'Box operations'}
B1 = {NewBox 1}
B2 = {NewBox 2}
{BoxExchange B1 V1old 7}
{BoxExchange B2 V2old 99}
{Test V1old '==' 1}
{Test V2old '==' 2}
declare V1x V2x
{BoxExchange B1 V1x 88}
{BoxExchange B2 V2x 333}
{Test V2x '==' 99}
{Test V1x '==' 7}
{Test {BoxAccess B1} '==' 88}
{Test {BoxAccess B2} '==' 333}
{Test {BoxAccess B2} '==' 333}
{Test {BoxAccess B1} '==' 88}
{BoxAssign B1 4}
{Test {BoxAccess B1} '==' 4}
{Test {BoxAccess B2} '==' 333}
{BoxAssign B1 asymbolliteral}
{Test {BoxAccess B1} '==' asymbolliteral}
declare
X=1 Y=2 Z=3
B={NewBox Z}
{StartTesting 'Some equations'}
{Test {BoxAccess {NewBox X}} '==' X}
{BoxAssign B Y}
{Test {BoxAccess B} '==' Y}
```

Figure 1: Testing code for Exercise 1 on the previous page.

You should use the `NewPortObject.oz` file, given in the book and supplied with the test cases for this homework, in your solution. (Hint: represent a Box with a port object, have `NewBox` return a port, and have the other functions send messages to the port. The state of the port object will be the Box's value.)

You are *not* allowed to use cells in your solution!

2. (20 points) [Concepts] [UseModels]

   Using Cells, but without using the message passing primitives `NewPort` and `Send`, define in Oz an ADT `PortAsCell`, which acts like the built-in port type, but is represented as a Cell. (For more about the imperative model and cells, look back at Section 1.12 of the textbook or forward to chapter 6.) The `PortAsCell` ADT has two operations:

   ```
   MyNewPort: <fun <$ Stream>: PortAsCell>
   MySend: <proc <$ PortAsCell Value>>,
   ```

   which are intended to act like `NewPort` and `Send`. That is, the function `MyNewPort` takes an unbound store variable, and returns a `PortAsCell`, which is a Cell that we want to act like a Port. The procedure `MySend` takes such a `PortAsCell` and a Value and adds the Value to the corresponding stream. In other words, the idea behind the ADT `PortAsCell` is that it should act like the built-in Port ADT of Oz, but be represented using Cells.

   For this problem, don't worry about the improper uses of the stream argument to `MyNewPort` (see Exercise 3). Also for the moment, don't worry about potential race conditions when multiple threads are used.

   Hint: look at the semantics for `NewPort` and `Send` in section 5.1 of the textbook, which shows how these work in terms of the mutable memory. Mutable memory is like a collection of cells. Think of the port name as being represented by the cell's identity. The cell holds the unbound store variable that is the end of the list. Note how the semantics of `Send` manipulates the mutable memory and works with new unbound store variables.

   Your code should pass the tests shown in Figure 2 on the next page.

3. (20 points; extra credit) [Concepts] [UseModels]

   Using read-only views (see Sections 3.7.5 and 13.1.14 of our textbook [RH04]), fix your solution to Exercise 2 so that your code has the same behavior for improper uses of the stream argument passed to `MyNewPort` as does Oz's built-in `NewPort` primitive.

   You must show by writing your own tests that your code has the required same behavior for such uses. Testing is up to you and an important part of this exercise.

4. [Concepts] [EvaluateModels]

   This problem concerns the expressive power of the message passing model in comparison to the imperative model.

   (a) (5 points) Can the Box ADT of Exercise 1 on page 1 do everything that a Cell can do in Oz? Briefly explain.

   (b) (5 points) Can the PortAsCell ADT of Exercise 2 do everything that a Cell can do in a sequential Oz program? Briefly explain.

   (c) (10 points) Does your implementation of the PortAsCell ADT of Exercise 2 have any potential race conditions when used in an Oz program with multiple threads? That is, is it possible that `MyNewPort` or `MySend` act differently than `NewPort` and `Send` when there are multiple threads? Briefly explain your answer.

   (d) (5 points) Is there any significant difference in expressive power between adding Cells or adding `NewPort` and `Send` to the kernel language?

```
% $Id: PortAsCellTest.oz,v 1.2 2007/11/18 14:09:59 leavens Exp leavens $
\insert 'PortAsCell.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'MyNewPort'}
% Simulating basic semantics of NewPort and Send
declare Strm Port in
Port = {MyNewPort Strm}
{StartTesting 'MySend'}
{MySend Port 3}
{MySend Port 4}
% Must use List.take, otherwise Test suspends...
{Test {List.take Strm 2} '==' [3 4]}
{MySend Port 5}
{MySend Port 6}
{Test {List.take Strm 4} '==' [3 4 5 6]}

{StartTesting 'MyNewPort second part'}
declare S2 P2 U1 U2 in
P2 = {MyNewPort S2}
{StartTesting 'MySend second part'}
{MySend P2 7}
{MySend P2 unit}
{MySend P2 true}
{MySend P2 U1}
{MySend P2 hmmm(x:U2)}
U1 = 4020
{Test {List.take S2 5} '==' [7 unit true 4020 hmmm(x:U2)]}
{Test {List.take Strm 4} '==' [3 4 5 6]}
```

Figure 2: Testing code for Exercise 2 on the preceding page.

5. (10 points) [EvaluateModels]

   Would it have been easier to use the message passing model to solve the square root approximation and differentiation exercises (numbers 13, 14, and 16) of homework 4? Briefly explain your answer. (You don't have to actually solve these problems with the message passing model.)

6. (10 points) [EvaluateModels]

   Suppose you are asked to program a simulation of an agent-based auction system for someone doing research in economics. This system consists of several independent agents, each of which must communicate with a central auction server to evaluate merchandise, place bids, and make payments.

   Among the programming models we studied this semester, What is the most restrictive (i.e., the least expressive or smallest) programming model that can practically be used program the overall structure of such a system? Briefly justify your answer.

Message Passing Programming

7. (60 points) [UseModels]

   Do part (a) of problem 3 in section 5.9 of the textbook [RH04] (Fault tolerance for the lift control system).

   You can get the code for the book's figures from the textbook's supplementary web site, which is: `http://www.info.ucl.ac.be/~pvr/ds/mitbook.html` or more directly from the book's supplementary web directory: `http://www.info.ucl.ac.be/~pvr/bookfigures/` or even more directly from WebCT.

   Note that in part (a) "when the floor is called" refers to `call` messages sent to a `Floor` port object.

   Feel free to make other changes to the code to make it more easy for you to understand and more sensible.

   Since deciding when the code works correctly is not easy, you are responsible for your own testing for this problem. In essence, you should set up a situation where a lift gets blocked on some floor, and make sure that the floor's timer is reset and that the blocked lift's schedule is distributed to other lifts and that the floors aren't calling the blocked lift. You can should add extra outputs to see whether the system is functioning as you intend.

   Now for some hints. To get started, overall it's useful to understand how the state diagrams (like Figure 5.7) relate to the code (like Figure 5.8's `Timer` class). Think about the design at the level of the state diagrams, and then make the corresponding changes to the code. You may find it useful to look at the diagrams to understand the code (and vice versa). Also don't be afraid to introduce new state (or parts of state) and new kinds of messages.

   To get started coding, look at the `Floor` component in Figure 5.10. When it's in the `doorsopen` state and it gets a `call` message is when the lift is being blocked at a floor. You'll have to change the code in that spot to start with. Get it to work reset the timer, perhaps by having the floor remember how many `stoptimer` messages it should wait for before telling the lift to close the doors. Then add to the `Lift` a way to get the schedule, and then figure out what component should ask for it and redistribute the schedule to other lifts.

   Don't wait until the last minute to start working on this problem.

   Be sure to mark, with comments, any changes you made to the code to solve the problem.

8. (60 points; extra credit) [UseModels]

   For extra credit, you can do one of parts (b)–(e) of problem 3 in section 5.9 of the textbook [RH04] (Fault tolerance for the lift control system).

   In part (b), you can design a wrapper port object that takes a `disable` message and then sends the `down` messages suggested; this wrapper can take port for the "linked" component as an argument, and send it the `down` message when the wrapper instance gets the `disable` message. That will allow testing.

9. (80 points) [Concepts] [UseModels] [MapToLanguages]

   Do problem 7 in section 5.9 of the textbook [RH04] (Erlang's receive as a control abstraction).

   We have provided a test script for this that is attached to the homework in the file hw5-tests.zip.

   Note that, according to the book's errata, when the `D` argument to `Mailbox.receive` has the form `T#E`, then `E` should be a *zero*-argument function.

   Some hints. Be sure to read section 5.7.3 in the textbook. Figure 5.21 in particular explains the semantics of `receive`, although you are not going to code that directly.

   In the problem, the name `Mailbox` is a variable identifier that is bound to a record containing three fields: `new`, `send`, and `receive`. Thus in outline your code should do something like the following.

```
declare
local
   % ...
   fun {New} ... end
   proc {MSend C M} ... end
   fun {MReceive C PGL D} ... end
in
   Mailbox = mailbox(new:New send:MSend receive:MReceive)
end
```

   Now `Mailbox.new` denotes the function `New`, since it does a record selection on the `Mailbox` record.

   In essence, your function `New` will return a port object (or something that contains a port object), which you will make by calling `NewPortObject`. Then `MSend` and `MReceive` will send messages to the port object passed as their first argument (or contained in their first argument). However, `MSend` and `MReceive` do very little themselves. In particular, `MReceive` can pass along its arguments `PGL` and `D` in a message, so that the port object (which has the appropriate state) can do the work.

   The port object should contain a list (or queue) of unprocessed messages in its state. It is helpful to think of the port object as having two states, one in which it is receiving (i.e., still processing a receive), and another in which it is not receiving. You can use a timer like the one in Figure 5.8 for timeouts, but you may need to modify it to suit this problem.

References

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