Exam 2010-12-15
Advanced Functional Programming

Uppsala University
Department of information technology

Sven-Olof Nyström

December 14, 2010

Please read the following instructions carefully:

- The exam is five hours.
- This is a closed book exam. No literature or notes are allowed.
- Use a separate sheet for each question, and write your name on each sheet. Leave a margin. Don't write anything important in the top-left corner.
- Please write clearly. Keep in mind that if I cannot read an answer, I cannot give any points for it.
- If something is unclear, try to make reasonable assumptions and write them down. I plan to come in around 10:00 to answer questions. Make sure that you have read the whole exam then.
- The last pages contain some simple program examples in Erlang, Common Lisp and Haskell.

Maximum points: 100. Normally, 50 points are required for a 3 (passing) grade, and 80 for a 5 (very good) grade.

Read the whole exam before you start. If you get stuck on one question, proceed to the next—the questions are not necessarily ordered in level of difficulty.

1. **Simple Cells (8p)**

   Implement an Erlang process (let's call such a process *a cell*) that accepts a message `{try_key, K, P}`, where `K` is some term and `P` is a pid.

   The process should store a key and a value, and send the value back to the pid if `K` is equal to the key, and send 0 otherwise.

   Write a simple test program that uses this process.

2. **Cells with children (18p)**

   Extend the solution of the previous question to allow a cell to have any number of children (these are also cells and implement the same interface).

   A cell process should also accept a message `{add_child, C}`, where `C` is another cell process.

   As previously, if the process receives the message `{try_key, K, P}`, and `K` is equal to the key of the process, the value of the process is sent back. If `K` is different form the key, the process asks the children the same question and sends back the sum of the answers.

   Example:

   Suppose a process has the key `foo` and the value `42`. Also assume it has four children, with the following pairs of keys and values:

   ```
   {bar, 1}    {bar, 10}    {bar, 100}    {foo, 99}
   ```

   When the process receives a try-key message with `K` equal to `foo`, it responds with the value 42. When it receives a message with `K` equal to `bar`, it responds with the value 111. If the `try_key` message is sent with any other key, 0 is returned, assuming that none of the children have any children.

   A cell process should not store any other information about its children other than its pids. If a child process takes time to respond, the cell process should still be able to handle other requests.

3. **For in CL (18p)**

   Write a for-macro in Common Lisp. Given a variable `i`, a lower bound `a` and an upper bound `b`, it should evaluate the body of the loop for each value of `i` in the range `a` to `b-1`. In other words,

   ```
   (for i a b
      expression)
   ```

   should evaluate the expression for each value of `i` from `a` to `b-1`. `a` and `b-1` can be any Common Lisp expression.

   For example,

   ```
   (for i 0 4
      (print i)))
   ```

   should print

```
0
1
2
3
```

and

```
(for i 0 4
   (for j 0 i
      (print (list i j))))
```

should print

```
(1 0)
(2 0)
(2 1)
(3 0)
(3 1)
(3 2)
```

The expressions a and b should only be evaluated *once* for one execution of the loop.

4. **Infinite product (18p)**

   Given two lists (that may be infinite), compute the list of all pairs of elements that can be constructed from elements of the two lists. Please avoid any use of indexing. Try to make the computation of each element of the list take constant time.

   Give types for the functions you define. Also make sure the function works correctly when one of the two lists is finite.

   The result should not contain any spurious duplicates; if each element of the two lists occurs only once in that list, the pair should also occur exactly once. (This holds for the example below).

   Examples. (Your solution is not required to produce the elements in exactly the same order.)

```
makepairs [1..] [100..]

[(1,100),(2,100),(1,101),(3,100),(1,102),(2,101),(1,103),(4,100),
 (1,104),(2,102),(1,105),(3,101),(1,106),(2,103),(1,107),(5,100),
 (1,108),(2,104),(1,109),(3,102), ...]
```

5. **Type classes (10p)**

   Explain how type classes in Haskell can be used to implement overloading.

   How would type classes be used when giving the type of a polymorphic function that needs to compare the polymorphic values for equality (such as the `member` function)?

6. **Continuation (18p)**

A *sub-sequence* of a list is a list containing some elements of the list, in the same order. For example, the sub-sequences of the list `[1,9,8,9]` are the lists

`[1,9,8,9]`, `[1,9,8]`, `[1,9,9]`, `[1,8,9]`, `[9,8,9]`, `[1,9]`, `[1,8]`, `[9,8]`, `[9,9]`, `[8,9]` `[1]`,`[9]`,`[8]` and `[]`

Use continuations to write a function `find_sum(L, N, C)` that takes a list `L` of integers and an integer `N` and calls `C` for all sub-sequences of `L` whose sum is equal to `N`.

Examples: Given `L=[3,1,4,1,5,9,2]`, `N=7`, and `C` some function that prints its argument, the result should be the following lists:

```
[3,1,1,2]
[3,4]
[1,4,2]
[1,1,5]
[4,1,2]
[5,2]
```

Given the same list, but `N=13`, the result should be

```
[3,1,4,5]
[3,1,9]
[3,4,1,5]
[3,1,9]
[1,4,1,5,2]
[1,1,9,2]
[4,9]
```

7. **Anonymous interpretation (10p)**

Describe how to implement anonymous functions in an interpreter.

Good Luck!

Sven-Olof

```
-module(counter).

-export([start/0, loop/1, increment/1, value/1, stop/1]).

start() ->
    spawn(counter, loop, [0]).

increment(Counter) ->
    Counter ! increment.

value(Counter) ->
    Counter ! {self(), value},
    receive
{Counter, Value} ->
    Value
    end.

stop(Counter) ->
    Counter ! stop.

loop(Val) ->
    receive
increment ->
    loop(Val+ 1);
{From, value} ->
    From ! {self(), Val},
    loop(Val);
stop ->
    true;
_ ->
    loop(Val)
    end.
```

```
;; Factorial written in a rather clumsy imperative style.

(defun fac1 (n)
  (let ((p 1))
    (block my-loop
      (loop
         (when (= n 0)
           (return-from my-loop p))
         (setf p (* p n))
         (setf n (- n 1))))))

;; A slightly less clumsy imperative factorial. (dotimes (i n) ...)
;; iterates over the values 0 .. n-1.

(defun fac2 (n)
  (let ((p 1))
    (dotimes (i n)
      (setf p (* p (+ i 1))))
    p))
```

```lisp
;; Append two lists

(defun appendx (x y)
  (cond
    ((null x) y)
    (t (cons (car x) (appendx (cdr x) y)))))

;; Efficient list reversal using a help function.

(defun myrev (l)
  (labels
      ((rev-help (l acc)
         (cond
           ((null l) acc)
           (t (rev-help (cdr l) (cons (car l) acc))))))
    (rev-help l nil)))

;; A simple macro.

(defmacro my-if (condition then else)
  `(cond (,condition ,then)
         (t ,else)))
```

```haskell
fak :: Integer -> Integer
fak 0 = 1
fak n = n * fak (n-1)

quicksort :: Ord a => [a] -> [a]
quicksort []     =  []
quicksort (x:xs) =  quicksort [y | y <- xs, y<x ]
                 ++ [x]
                 ++ quicksort [y | y <- xs, y>=x]

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip  xs     ys    = []

fib :: [Integer]
fib = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]


class Weight a where
    weight :: a -> Integer

instance Weight Integer where
    weight x = x

instance Weight Int where
    weight x = toInteger(x)

instance Weight [a] where
    weight l = toInteger(length l)
```