ST Master Course on Advanced Functional Programming

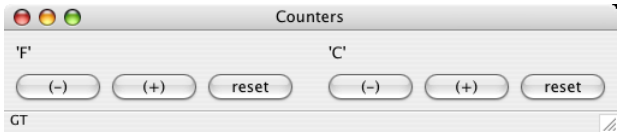
Friday, April 21, 2006 (9:00-12:00)

The exam consists of 5 open questions: the maximum number of points for each question is given (100 points in total). Give short and precise answers. If a Haskell function is asked for, try to find an elegant solution. It is recommended to read all parts of a question before you provide an answer. You may consult course material during the test. Good luck!

---

1 Counter Panel (15 points)

A *counter panel* is a panel that contains a text label showing a value, and a decrement, an increment, and a reset button. Such a panel is polymorphic in the value it controls, as long as we can *show* and *compare* values of that type. Furthermore, we use the *Enum* type class to implement the decrement and increment operations: *pred* (for decrement) and *succ* (for increment) both have type $Enum\ a \Rightarrow a \to a$. The following frame contains two counter panels for characters:



The status bar displays GT, which is the result of comparing the two current values (because `'F' > 'C'`). The value of a counter panel is restricted to a minimum and a maximum value. For this, we introduce another type class:

```
class (Show a, Ord a, Enum a) ⇒ MinMax a where
  minValue :: a
  maxValue :: a

instance MinMax Char where
  minValue = 'A'
  maxValue = 'Z'
```

The data type *Control* (from the second lab assignment) is used to implement the observer/observable pattern.

```
data Control a = C{model :: IORef a, observers :: IORef [IO ()]}

createControl :: a → IO (Control a)
createControl a =
  do m   ← newIORef a
     obs ← newIORef []
     return (C m obs)

getValue :: Control a → IO a
getValue = readIORef ∘ model

setValue :: Control a → a → IO ()
setValue ctrl a =
  do writeIORef (model ctrl) a
     readIORef (observers ctrl) ≫= sequence_
```

```
addObserver :: Control a → IO () → IO ()
addObserver ctrl callback =
  modifyIORef (observers ctrl) (callback:) ≫ callback
```

Now we can define the function *main*, which creates a status bar, two counter panels, and puts these into a frame.

```
main :: IO ()
main = start $
  do f       ← frame [text := "Counters"]
     status ← statusField []

     (cp1, ctrl1 :: Control Char) ← counterPanel f
     (cp2, ctrl2 :: Control Char) ← counterPanel f

     let callback c = addObserver c (updateBar status ctrl1 ctrl2)
     mapM_ callback [ctrl1, ctrl2]

     set f [statusbar := [status], layout := row 10 [widget cp1, widget cp2]]
```

Two functions used by *main* are not yet defined: their type signatures are:

```
updateBar    :: MinMax a ⇒ StatusField → Control a → Control a → IO ()
counterPanel :: MinMax t ⇒ Window a → IO (Panel (), Control t)
```

a) Give a definition for *updateBar* such that the status bar reflects the comparison of the two counter characters. Use *compare* :: *Ord a* ⇒ *a* → *a* → *Ordering* for comparing the *Char*s.

b) Define the function *counterPanel* such that:

- A new control and a new panel are created. Use *minValue* as the initial value.
- The panel should contain one text label and three buttons. The layout of the panel should resemble the layout of the screenshot.
- Implement the *command* events for the three buttons. Make sure that the value remains between *minValue* and *maxValue* at all time. The reset button should set the value to *minValue*.
- The text label showing the counter panel's value should change whenever the value of its control changes.

c) Suppose that we want to *share* the value of the two counters: pressing the (+) button of the left counter also increments the value displayed by the right counter (and vice versa). Describe how the program should be changed (code is not required).

2 List Monad (25 POINTS)

The following list comprehension generates an infinite list containing infinite lists:

```
squareList :: [[Int]]
squareList = [[sq, sq * 2 ..] | i ← [1 ..], let sq = i * i]
```

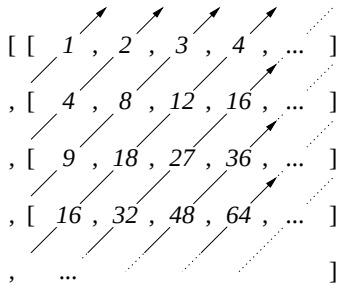
This list of lists could be visualized as follows:

$$(1:2:3:4:\ldots)\ :\ (4:8:12:16:\ldots)\ :\ (9:18:27:36:\ldots)\ :\ (16:32:48:64:\ldots)\ :\ \ldots$$

We have the *Prelude* function *concat* at our disposal to flatten this list. However, *concat squareList* will *only* return elements from the first list. The function *join*, defined below, has *concat*'s type, but takes elements in a diagonal fashion:

```
join :: [[a]] → [a]
join = rec []

  where
    rec [] []  = []
    rec as bs =
      let notEmpty = not ∘ null
          (hd, tl)    = splitAt 1 bs
          rest        = hd ++ map tail as
      in map head as ++ rec (filter notEmpty rest) tl
```



Indeed, the expression *take* 10 (*join squareList*) evaluates to $[1, 4, 2, 9, 8, 3, 16, 18, 12, 4]$.

a) Explain in your own words why the potentially dangerous functions *head* and *tail* in *join*'s definition are safe (no pattern match failures).

b) We continue with the introduction of a wrapper data type for a normal Haskell list:

```
newtype List a = List{ listify :: [a] } deriving Eq
```

This brings into scope the unwrapper function *listify* of type $List\ a \to [a]$. Make *List* an instance of the *Functor* type class:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

c) Give a point-free definition for $joinList :: List\ (List\ a) \to List\ a$, which uses the function *join* to combine lists. There will be a small penalty for a definition that is not point-free.

d) Having *join* and *map* (or actually, *fmap*) defined makes it easy to define the monadic ($\gg=$) operator. Make *List* an instance of the *Monad* type class. Of course you may reuse code defined earlier.

e) Remember the three algebraic laws for monads:

$$return\ x \gg= f \quad \equiv \quad f\ x \qquad \text{(LEFT UNIT)}$$
$$m \gg= return \quad \equiv \quad m \qquad \text{(RIGHT UNIT)}$$
$$m_1 \gg= (\lambda x \to m_2 \gg= (\lambda y \to m_3)) \quad \equiv \quad (m_1 \gg= (\lambda x \to m_2)) \gg= (\lambda y \to m_3) \qquad \text{(BIND)}$$
assuming that x does not appear free in m_3

Do the monad laws hold for the instance defined at **d)**? If you think they hold, give a short motivation (no formal proof is required). Otherwise, give a counter-example.

f) Define three *QuickCheck* properties to check the monad laws for your *List* instance. You may assume that a random data generator for *List*s is provided.

g) Suppose we want to have a monad transformer for *List*. Give a suitable type definition for this transformer. You do not have to give the instance declarations.

3 Operational Semantics (20 points)

We will study the behavior of the following function:

$f :: (Int, (Int, Int)) \to [Int]$
$f\ (a, (b, c)) = 0\ :\ a\ :\ f\ (c, (a, b))$

a) Consider the result of $f\ (1 + 2 + 3, (0, 0))$. Whether or not the subexpression $1 + 2 + 3$ is evaluated and reduced to 6 depends on the context of the expression. Describe two situations: one in which the reduction takes place, and one in which the subexpression is not evaluated.

b) The functions g, h, and k are variations on f:

$g\ {\sim}(a, (b, c)) = 0\ :\ a\ :\ g\ (c, (a, b))$
$h\ (a, {\sim}(b, c)) = 0\ :\ a\ :\ h\ (c, (a, b))$
$k\ (a, {\sim}(b, c)) = 0\ :\ a\ :\ seq\ b\ (k\ (c, (a, b)))$

Remember that the expression $x\ `seq`\ y$ evaluates x to weak head normal form (WHNF) and then returns y. Indicate as precise as possible at which point the evaluation of the expression $show\ \$\ f\ (1, \bot)$ diverges and returns $\bot$. Answer the same question for g, h, and k when used instead of f.

c) Evaluating *length xs* has one of the following outcomes (depending on *xs*):

- The length of the list is returned.
  Example: $length\ [1\,..\,10]$.
- An exception is thrown, or the computation is non-terminating.
  Examples: $length\ (head\ [])$ and $length\ [1\,..]$.

What is the outcome of evaluating $length\ \$\ f\ (1, \bot)$? Answering with "the length" or "an exception" suffices. Answer the same question for g, h, and k when used instead of f.

d) Write a function

$seqTriple :: (a, (b, c)) \to (a, (b, c))$

that forces evaluation to WHNF of all three components.

4 Generalized Algebraic Data Types (20 points)

Consider the following data type declarations:

```
data Succ n
data Zero

data LengthList n a where
  Cons :: a → LengthList n a → LengthList (Succ n) a
  Nil  :: LengthList Zero a
```

Be careful: *Succ* and *Zero* are types, not values. The data types *Succ* and *Zero* are empty: no value of such a type exists (except for $\bot$). *LengthList*'s first type argument is a phantom type, which we use for encoding the length of the list using *Succ* and *Zero*.

a) What type is inferred for the expression $Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))$?

b) Assume that we want *Append*, which combines two lists, to be *LengthList*'s third constructor function. Because the length of *xs* (n_1) and the length of *ys* (n_2) together determine the length of *Append xs ys* ($n = n_1 + n_2$), we need a type class for "adding two types":

$$\textbf{class } Add\ n_1\ n_2\ n \mid n_1\ n_2 \to n$$

We extend *LengthList*'s GADT with the following constructor function:

$$Append :: Add\ n_1\ n_2\ n \Rightarrow LengthList\ n_1\ a \to LengthList\ n_2\ a \to LengthList\ n\ a$$

Explain the purpose of the functional dependency in the type class *Add*.

c) Give instance declarations such that we can add *Succ*s and *Zero*s on the type-level. Hint: try to find an inductive definition on the value-level first.

d) Consider the following two conversion functions:

$$\begin{aligned} &withLength &&:: [a] \to LengthList\ n\ a \\ &withoutLength &&:: LengthList\ n\ a \to [a] \end{aligned}$$

Give a definition for these two functions: also take the constructor function *Append* into account. If you feel that one (or two) of these functions cannot be defined, then motivate your judgement instead.

e) The *Prelude*'s function *head* throws an exception when applied to the empty list. We could provide a similar function for *LengthList*. Because the length of the list is statically known (it is part of the type), we can define a function *safeHead* that returns the first element for a non-empty list, and results in a *type error* when applied to *Nil* or *Append Nil Nil*. Give a type signature and a definition for this function.

5 Stream Functions (20 POINTS)

Recall the arrow of *stream functions*:

$$\textbf{newtype } SF\ a\ b = SF\{\, runSF :: [a] \to [b]\,\}$$

This arrow supports all operations of the *Arrow* and the *ArrowLoop* type classes. For your convenience, here is a list of available combinators for composing stream functions:

$$\begin{aligned} &arr &&:: Arrow\ arr \Rightarrow (a \to b) \to arr\ a\ b \\ &(>>>) &&:: Arrow\ arr \Rightarrow arr\ a\ b \to arr\ b\ c \to arr\ a\ c \\ &first &&:: Arrow\ arr \Rightarrow arr\ a\ b \to arr\ (a, c)\ (b, c) \\ &second &&:: Arrow\ arr \Rightarrow arr\ b\ c \to arr\ (a, b)\ (a, c) \\ &(***) &&:: Arrow\ arr \Rightarrow arr\ a\ c \to arr\ b\ d \to arr\ (a, b)\ (c, d) \\ &(\&\&\&) &&:: Arrow\ arr \Rightarrow arr\ a\ b \to arr\ a\ c \to arr\ a\ (b, c) \end{aligned}$$

$$loop :: ArrowLoop\ arr \Rightarrow arr\ (a, c)\ (b, c) \to arr\ a\ b$$

In addition to these arrow combinators, we provide two more utility functions: *delay* for delaying a stream by one element, and *plus* for adding two values of type *Int*.

$$\begin{aligned} &delay :: a \to SF\ a\ a \\ &delay\ x = SF\ (x:) \end{aligned}$$

$$\begin{aligned} &plus :: Arrow\ arr \Rightarrow arr\ (Int, Int)\ Int \\ &plus = arr\ (uncurry\ (+)) \end{aligned}$$

These are all the ingredients needed to construct the cyclic machine depicted in Figure 1. In Haskell, this machine is given the type *SF Int Int*.

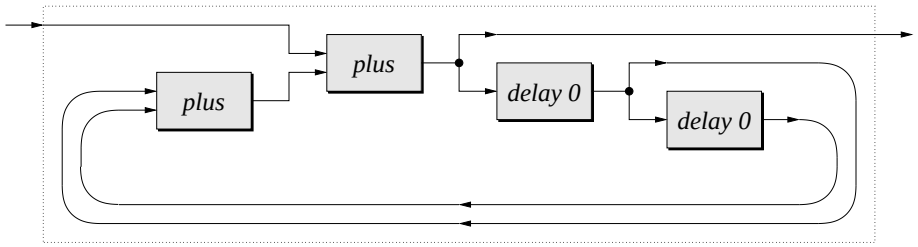


Figure 1: A stream function with two *plus* and two *delay* arrows.

a) Suppose we supply the stream (*repeat* 1) to this machine. Then what are the first five elements of the output stream? Indicate briefly how you arrived at your answer.

b) Define the machine of Figure 1 using the arrow combinators.

c) By choosing the right input stream for our machine, we can compute the sequence of Fibonacci numbers:

$$0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : ...$$

Define the infinite list

$$\mathit{fibs} :: [\mathit{Int}]$$

that returns all Fibonacci numbers by running the stream function you have defined for question **b)**.

d) The sequence of Fibonacci numbers can be computed by adding the two previous Fibonacci numbers. This corresponds nicely with the fact that our machine contains two *delay* and two *plus* arrows. We will generalize our machine. Write a recursive function *delays* that takes an integer n and then composes n *delay* components similar to the machine in Figure 1. This function should have the following type:

$$\mathit{delays} :: \mathit{Int} \rightarrow \mathit{SF}\ \mathit{Int}\ [\mathit{Int}]$$

All lists in the output stream should have length n ("the number of wires used for feedback"). Hint: this observation may help you to define the base case for *delays*.

e) Define the function

$$\mathit{genMachine} :: \mathit{Int} \rightarrow \mathit{SF}\ \mathit{Int}\ \mathit{Int}$$

that generalizes the stream function shown in Figure 1.