# ST Master Course on Advanced Functional Programming

Wednesday, April 18, 2007 (9:00-12:00)

*This exam consists of 5 open questions: the maximum number of points for each question is given (100 points in total, plus an additional 5 bonus points). 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 Laziness and Strictness (10 points)

Let *prime1000* be a computation that yields the 1000th prime number (a value of type *Int*). Obviously, evaluating *prime1000* takes some time. Furthermore, we define a helper data type with strictness annotations:

**data** $X = X\ !Int\ !Bool$

Suppose that we want to evaluate the following expressions. Indicate for each expression whether *prime1000* is **not evaluated at all**, or that *prime1000* has to be **fully evaluated**.

**a)** **let** $f\ (a, b) = a$ **in** $f\ (True, prime1000)$

**b)** **let** $f\ [x] = True$ **in** $f\ [prime1000]$

**c)** **let** $f\ (X\ a\ b) = b$ **in** $f\ (X\ prime1000\ True)$

**d)** **let** $f\ {\sim}(X\ a\ b) = b$ **in** $f\ (X\ (prime1000 + prime1000)\ True)$

**e)** **let** $f\ xs@(\_ : \_) = length\ xs$ **in** $f\ (X\ (prime1000 + prime1000)\ True : [\,])$

## 2 Tracing Arithmetic Expressions (20 points)

We will use the *Kleisli* arrow to trace the evaluation of simple arithmetic expressions. We begin with an example trace:

```
*Main> runTerm $ (3 + 4) * (2 + input)
3 + 4 = 7
? 1
2 + 1 = 3
7 * 3 = 21
result: 21
```

The variable *input* will prompt the user to enter a value: in the given example trace, the user provided the value 1 (third line). Each step in the evaluation of the term is reported, and so is the final result. The definition of the *Kleisli* arrow and the class declaration for *Arrow* (that comes with `ghc-6.6`) can be found below:

$$\textbf{newtype}\ Kleisli\ m\ a\ b = Kleisli\{runKleisli :: a \to m\ b\}$$

$$
\begin{array}{l}
\textbf{class}\ Arrow\ a\ \textbf{where} \\
\quad arr \quad\ \, :: (b \to c) \to a\ b\ c \\
\quad pure \quad :: (b \to c) \to a\ b\ c \\
\quad (\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d \\
\quad first \quad\ :: a\ b\ c \to a\ (b, d)\ (c, d) \\
\quad second :: a\ b\ c \to a\ (d, b)\ (d, c) \\
\quad (***) \ \ :: a\ b\ c \to a\ b'\ c' \to a\ (b, b')\ (c, c') \\
\quad (\&\&\&) :: a\ b\ c \to a\ b\ c' \to a\ b\ (c, c')
\end{array}
$$

We first define a type synonym for the arithmetic expressions that we want to trace:

$$\textbf{type}\ Term = Kleisli\ IO\ ()\ Integer$$

The side-effects take place in the *IO* monad, the term does not depend on any input (hence the type ()), and the output is of type *Integer*.

**a)** Define the function *con* that turns an *Integer* into a *Term* (without any side-effect taking place):

$$con :: Integer \to Term$$

**b)** Define the function *input* that asks the user for some input:

$$input :: Term$$

You may want to use $getLine :: IO\ String$ for this.

**c)** Next, we will define some binary operators to combine two values. First, we introduce a type synonym for binary operators:

$$\textbf{type}\ BinOp = Kleisli\ IO\ (Integer, Integer)\ Integer$$

Define binary operators for addition and multiplication:

$$plus, times :: BinOp$$

Besides delivering a value, these two operators will have a side-effect: a simple equation reports the two operands, the operator, and the result to the user.

**d)** Implement the function *apply* that takes a binary operator and two operands and yields a new *Term*:

$$apply :: BinOp \to Term \to Term \to Term$$

**e)** With the definitions of *con*, *apply*, *plus*, and *times* available, we make *Term* an instance of the *Num* type class:

$$
\begin{array}{l}
\textbf{instance}\ Eq\quad Term \\
\textbf{instance}\ Show\ Term \\
\\
\textbf{instance}\ Num\ Term\ \textbf{where} \\
\quad fromInteger = con \\
\quad (+) \qquad\quad\ = apply\ plus \\
\quad (*) \qquad\quad\ = apply\ times
\end{array}
$$

The instance declarations above are only defined for syntactic convenience. However, these declarations do not conform to the Haskell 98 standard (but with `-fglasgow-exts` enabled, they are accepted). Why do they violate the Haskell 98 standard?

**f)** Define the function *runTerm* that runs the *Kleisli* arrow and reports the final result:

$$runTerm :: Term \to IO\ ()$$

# 3 Success or Failure? (25 points)

With the data type *Step* we can encode success, failure, and a sequence of success/failure steps:

$$\textbf{data}\ Step\ a = Success\ a \mid Fail \mid Steps\ [Step\ a]$$

**a)** Make *Step* an instance of the *Functor* type class. This type class is defined in the standard *Prelude* by:

$$\textbf{class}\ Functor\ f\ \textbf{where}$$
$$fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

**b)** Write the monadic *join* function for the *Step* data type:

$$join :: Step\ (Step\ a) \rightarrow Step\ a$$

**c)** Turn the *Step* data type into a *Monad*. Your instance declaration should respect the three monad laws (but you don't have to prove this).

**d)** With *Step* being a *Monad*, we can now use Haskell's **do** notation:

$$m :: Step\ (Int, Int)$$
$$m = \textbf{do}\ a \leftarrow Success\ 1$$
$$b \leftarrow Steps\ [Fail, Success\ 2]$$
$$return\ (a, b)$$

Give the value of $m$ in terms of the three constructor functions of *Step*.

**e)** The following law should hold for the *Step* monad (we use *fmap* for the monadic *map* to avoid confusion with the specialized implementation for lists from the *Prelude*):

$$fmap\ f \circ fmap\ g \equiv fmap\ (f \circ g)$$

Prove that the instance declaration you provided for **a)** respects this law. (If the law is violated, then change your instance declaration.)

**f)** We want a *Polish* (linear) representation for the data types *Step*, list, and *Int*. The following *Polish* type definitions are given:

$$\textbf{data}\ StepP\ a\ c = \ ...$$
$$\textbf{data}\ ListP\ a\ c = ConsP\ (a\ (ListP\ a\ c)) \mid NilP\ c$$
$$\textbf{data}\ IntP\ c \quad = IntP\ Int\ c$$

$$\textbf{type}\ StepIntP = StepP\ IntP$$

The extra type argument $c$ is the continuation (or the "future"). Complete the definition for *StepP*, which should still have three constructor functions.

**g)** Give kind signatures for *StepP*, *ListP*, and *IntP*.

**h)** Consider the following definition:

$$steps :: Step\ Int$$
$$steps = Steps\ [Success\ 1, Steps\ [Fail, Success\ 2]]$$

Define the value $stepsP :: StepIntP\ ()$ which is the *Polish* representation of *steps*.

**i)** (Bonus: 5 points) Implement the function

$$collectInts :: StepIntP\ c \rightarrow ([Int], c)$$

which collects all values of type *Int* and returns these in a list paired with the rest of the continuation.

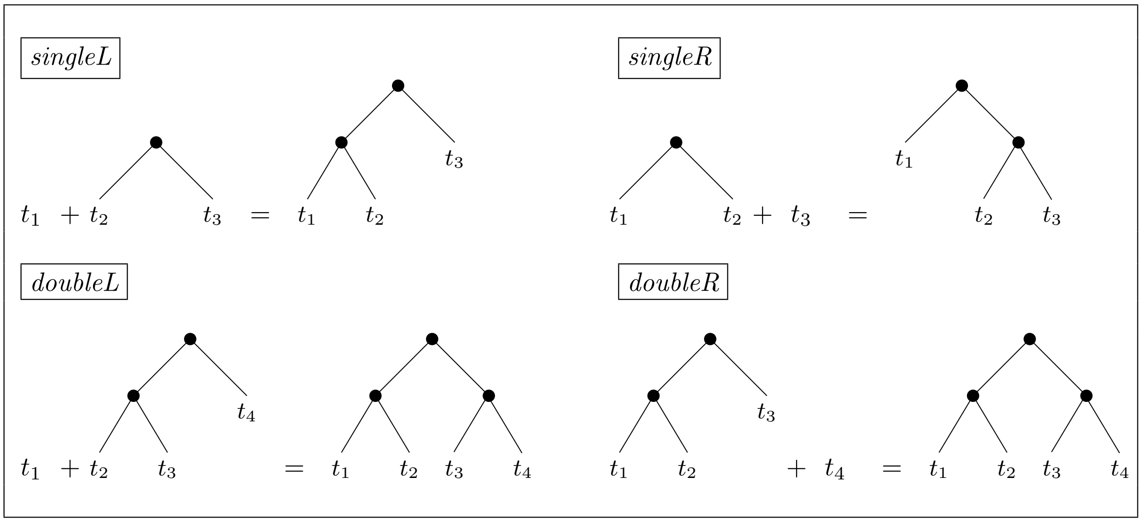



Figure 1: Four rotation functions

## 4 Balanced Trees (25 points)

We use the following data type for balanced trees:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Bin}\ !\mathit{Int}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \mid \mathit{Leaf}\ a$$

In the implementation we respect the following two invariants:

- At each *Bin* constructor, we store the number of values in the two subtrees. Values are only stored in the leafs.
- All trees that we construct are balanced. We say that a tree is balanced if (and only if) for each internal node it holds that

  $$\mathit{size}\ l \leqslant \mathit{size}\ r * 2 \quad \wedge \quad \mathit{size}\ r \leqslant \mathit{size}\ l * 2$$

  where $l$ and $r$ are the node's two subtrees, and *size* returns the number of values stored in a tree.

The relative order in which the values of a tree are stored is considered to be relevant: values and/or subtrees are not supposed to be swapped.

**a)** Define the function

$$\mathit{size} :: \mathit{Tree}\ a \rightarrow \mathit{Int}$$

that returns the number of values stored in a tree. This should be a constant time operation.

**b)** We need four rotation functions to keep our trees balanced: these functions are depicted in Figure 1. All these functions take two balanced trees and return a balanced tree. Although the shape changes, observe that the relative order of the values does not change. Define the four rotation functions:

$$\mathit{singleL}, \mathit{singleR}, \mathit{doubleL}, \mathit{doubleR} :: \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a$$

**Hint:** To make sure that the tree that is returned by a rotation function is balanced, you may already use the smart constructor *bin* which we will define next.

**c)** Define the smart constructor *bin* that combines two (balanced) trees.

$$bin :: Tree\ a \rightarrow Tree\ a \rightarrow Tree\ a$$

**Hint:** Consider five possible scenarios. Either the two trees can be combined without any rotation, or one of the four rotation functions should be used. It is not a problem if the smart constructor and the rotation functions are mutually recursive.

**d)** For the last part we use *QuickCheck* to validate our implementation. Make the *Tree* data type an instance of the *Arbitrary* type class (provided that we also have an *Arbitrary* instance for the type of the elements in the tree). Also define the *coarbitrary* member function. All trees that are randomly generated should respect the two invariants. Make sure that the generation of random trees terminates.

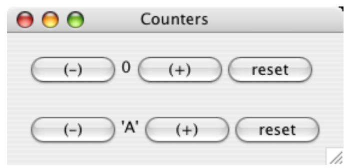
**e)** Write *QuickCheck* properties for the two invariants of the *Tree* data type. Also write a function

$$checkAll :: IO\ ()$$

that *quickly checks* all properties you have defined.

## 5 Counters in wxHaskell (20 points)

We will program a *wxHaskell* application that contains two counters: the first counter has a value of type *Int*, the second a value of type *Char*. Each counter comes with an increment button, a decrement button, and a reset button. The intended layout of the application is shown below:




The following data type is used for implementing a counter:

$$\mathbf{data}\ \mathit{ValueDisplay}\ a = \mathit{VD}\ (\mathit{StaticText}\ ())\ (\mathit{IORef}\ a)$$

A *ValueDisplay* consists of a widget displaying the value (we use *StaticText* for this) and a mutable reference holding the current value.

**a)** Implement a function that takes a parent window and an initial value and constructs a *ValueDisplay*. This function should have the following type:

$$\mathit{valueDisplay} :: \mathit{Show}\ a \Rightarrow \mathit{Window}\ w \rightarrow a \rightarrow \mathit{IO}\ (\mathit{ValueDisplay}\ a)$$

**b)** Write a function for changing the value of a *ValueDisplay*:

$$\mathit{changeValue} :: \mathit{Show}\ a \Rightarrow \mathit{ValueDisplay}\ a \rightarrow (a \rightarrow a) \rightarrow \mathit{IO}\ ()$$

Of course, any update must be reflected in the *StaticText* widget.

**c)** The following type class is defined in the wxHaskell library:

$$\begin{array}{l}\mathbf{class}\ \mathit{Widget}\ w\ \mathbf{where} \\ \quad \mathit{widget} :: w \rightarrow \mathit{Layout}\end{array}$$

Make *ValueDisplay* an instance of the *Widget* type class.

**d)** The function *displayPanel* constructs several widgets that are part of a counter:

$$\mathit{displayPanel} :: (\mathit{Enum}\ a, \mathit{Show}\ a) \Rightarrow \mathit{Window}\ w \rightarrow a \rightarrow \mathit{IO}\ (\mathit{Panel}\ ())$$

This function takes a parent window and an initial value for the counter, and then constructs and returns a new panel. This panel should contain three buttons (increment, decrement, and reset) as well as a *ValueDisplay*. Define *displayPanel*: also implement the event handlers of the buttons and the panel's layout. Note that we use the *Enum* type class for incrementing and decrementing the value.

**e)** Implement the function $\mathit{main} :: \mathit{IO}\ ()$ to start the GUI. The application should have a title and two counters that are initially set to 0 and `'A'`, as suggested by the screenshot above.