ST Master Course on Advanced Functional Programming

Tuesday, July 5, 2005 (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, and provide a type signature. It is recommended to read all parts of a question before you provide an answer. Good luck!

1 Typed Controls (25 points)

The *wxHaskell* GUI library offers a number of controls, including text entries. A text entry can be created with the function

$$textEntry :: Window\ a \rightarrow [Prop\ (TextCtrl\ ())] \rightarrow IO\ (TextCtrl\ ())$$

and it uses the attribute $text :: Textual\ w \Rightarrow Attr\ w\ String$ to access the value of the input field. A disadvantage of this *text* attribute is that its value is always a *String*, even if you want the user to input a value of type *Int*. To remedy this problem, you have to implement a *typed* text entry on top of the existing text control.

a) Introduce a new type constant *TypedEntry*: use either a **data** or a **type** declaration. This constant is to be used as *TypedEntry t a*, where *t* denotes the type of the value (for instance, *Int*), and *a* is used to model inheritance. Make *TypedEntry* a subtype of *TextCtrl*: all functions that expect a value of type *TextCtrl a* can be passed a value of type *TypedEntry t a*.

```
data CTypedEntry t w
type TypedEntry   t w = TextCtrl (CTypedEntry t w)
```

b) Next, we introduce a multi-parameter type class for controls that contain a value of a certain type.

```
class TypedValue t w | w → t where
  typedValue :: Attr w (Maybe t)
```

Explain the meaning of the functional dependency $w \rightarrow t$, and why this is necessary.

The functional dependency states that the widget uniquely determines the type of the value contained by the widget. For instance, if the widget has type *TypedEntry Int* (), then the attribute *typedValue* for this widget must be of type *Maybe Int*. The functional dependency prevents type class ambiguities, and rules out certain (undesired) instance declarations for the type class *TypedValue*.

c) Of course we make *TypedEntry* an instance of the type class *TypedValue*. Assume that we use the *Show* and *Read* type classes for converting from and to a *String*. We introduce a new attribute *typedValue*.

```
instance (Show t, Read t) ⇒ TypedValue t (TypedEntry t a) where
  typedValue = newAttr "typedValue" getter setter
```

Give a definition for the *getter* and *setter* functions that are used in the instance declaration above. These two functions should have the following types:

```
getter :: Read t ⇒ TypedEntry t a → IO (Maybe t)
setter :: Show t ⇒ TypedEntry t a → Maybe t → IO ()
```

You may use the following helper function:

```
parseRead :: Read a ⇒ String → Maybe a
parseRead s = case reads s of
                [(a, "")] → Just a
                _         → Nothing
```

```
getter entry =
  do s ← get entry text
     return (parseRead s)

setter entry mValue =
  case mValue of
    Nothing    → return ()
    Just value → set entry [text := show value]
```

d) Finally, you are asked to implement the function *typedEntry* for constructing a *TypedEntry*.

```
typedEntry :: (Show t, Read t) ⇒
              Window a → [Prop (TypedEntry t ())] → IO (TypedEntry t ())
```

A default behavior of a *TypedEntry* should be that as soon as it gets/loses the focus, the content of the *TypedEntry* should be presented in red if it cannot be parsed (and in black otherwise). You will need the wxHaskell function *objectCast* for an unsafe *cast* between two objects (the superclass of all controls). The functions *objectCast* and *focus* have the following types:

```
objectCast :: Object a → Object b
focus      :: Reactive w ⇒ Event w (Bool → IO ())
```

```
typedEntry parent props =
  do normalEntry ← textEntry parent []
     valueEntry  ← return (objectCast normalEntry)
     set valueEntry (props ++ [on focus := onFocus valueEntry])
     return valueEntry

onFocus :: (Show x, Read x) ⇒ TypedEntry x () → Bool → IO ()
onFocus entry _ =
  do mValue ← get entry typedValue
     set entry [color := maybe red (const black) mValue]
     repaint entry
```

2 Monads and QuickCheck (25 POINTS)

a) Instead of defining a monad by *bind* (≫=) and *return*, we can also define it in terms of *map* (hereafter called *mmap* to avoid confusion with *map* from the *Prelude*), *join*, and *return*.

```
mmap :: Monad m ⇒ (a → b) → m a → m b
join :: Monad m ⇒ m (m a) → m a
```

Define *mmap* and *join* in terms of (≫=) and *return*, and define (≫=) in terms of *mmap*, *join*, and *return*.

```
mmap f xs = xs  ≫= (return ∘ f)
join xss     = xss ≫= id
xs ≫= f      = join (mmap f xs)
```

b) A *Snoc*-list is an alternative representation of a (normal) list: a non-empty *Snoc*-list consists of the last element of the list, and the rest.

```
data Snoc a = Nil | Snoc a :< a deriving (Show, Eq)
```

Define *listToSnoc*, such that *listToSnoc* [1 . . 5] returns

$((((Nil :< 1) :< 2) :< 3) :< 4) :< 5$

Use higher-order functions if possible.

```
listToSnoc :: [a] → Snoc a
listToSnoc = foldl (:<) Nil
```

c) Define the operator $(<\!\!+\!\!+) :: Snoc\ a \to Snoc\ a \to Snoc\ a$ for concatenating two *Snoc*-lists.

```
(<++) :: Snoc a → Snoc a → Snoc a
xs <++ Nil        = xs
xs <++ (ys :< a) = (xs <++ ys) :< a
```

d) Make *Snoc* an instance of the *Monad* type class such that the three monad laws are satisfied. Give a definition for the member functions (≫=) and *return*.

```
instance Monad Snoc where
  return a        = Nil :< a
  Nil       ≫= _ = Nil
  (s :< a) ≫= f = (s ≫= f) <++ (f a)
```

e) Give the type and the result of the function *test*.

```
test = do x ← listToSnoc [1 . . 5]
          return (x ∗ x)
```

The type of *test* is *Snoc Int*, or, more precise, $(Enum\ a, Num\ a) \Rightarrow Snoc\ a$. Evaluating *test* results in $((((Nil :< 1) :< 4) :< 9) :< 16) :< 25$.

f) Prove that the first two monad laws hold for a *Snoc*-list. Use equational reasoning.

$$\begin{array}{rcll} (return\ x) \gg\!= f & \equiv & f\ x & \text{(left-identity)} \\ m \gg\!= return & \equiv & m & \text{(right-identity)} \end{array}$$

For the following proofs, we use that $Nil \mathbin{<\!\!+\!\!+} xs = xs$ (follows from its definition).

Left-identity law:

$$\begin{aligned}
&return\ x \gg\!\!= f \\
&\quad = (Nil :< x) \gg\!\!= f && \text{-- definition } return \\
&\quad = (Nil \gg\!\!= f) \mathbin{<\!\!+\!\!+} (f\ x) && \text{-- definition } \gg\!\!= \\
&\quad = Nil \mathbin{<\!\!+\!\!+} f\ x && \text{-- definition } \gg\!\!= \\
&\quad = f\ x && \text{-- definition } \mathbin{<\!\!+\!\!+}
\end{aligned}$$

Right-identity law: proof by induction on the length of m

case 1: $|m|= 0$, then $m = Nil$.

$$\begin{aligned}
&m \gg\!\!= return \\
&\quad = Nil \gg\!\!= return && \text{-- definition } m \\
&\quad = Nil && \text{-- definition } \gg\!\!= \\
&\quad = m && \text{-- definition } m
\end{aligned}$$

case 2: $|m| > 0$, then $m = s <: x$ I.H.: $(s \gg\!\!= return) = s$

$$\begin{aligned}
&m \gg\!\!= return \\
&\quad = (s :< x) \gg\!\!= return && \text{-- definition } m \\
&\quad = (s \gg\!\!= return) \mathbin{<\!\!+\!\!+} (return\ x) && \text{-- definition } \gg\!\!= \\
&\quad = s \mathbin{<\!\!+\!\!+} (Nil :< x) && \text{-- I.H.} \\
&\quad = (s \mathbin{<\!\!+\!\!+} Nil) :< x && \text{-- definition } \mathbin{<\!\!+\!\!+} \\
&\quad = s :< x && \text{-- definition } \mathbin{<\!\!+\!\!+} \\
&\quad = m && \text{-- definition } m
\end{aligned}$$

g) Give a *QuickCheck* property to check that also the third monad law holds.

$$(m \gg\!\!= f) \gg\!\!= g \quad \equiv \quad m \gg\!\!= (\lambda x \to f\ x \gg\!\!= g) \qquad \text{(associativity)}$$

Indicate precisely how this property can be validated by the *QuickCheck* system, and make sure that *QuickCheck* is able to generate random *Snoc*-lists (with an appropriate distribution of test cases).

To generate random *Snoc*-lists, we slightly modify the instance declaration for *Arbitrary* $[a]$.

```
instance Arbitrary a ⇒ Arbitrary (Snoc a) where
  arbitrary = frequency
    [(1, return Nil)
    ,(4, liftM2 (:<) arbitrary arbitrary)
    ]
```

The third monad law can be checked by:

```
associativity :: Snoc Int → (Int → Snoc Int) → (Int → Snoc Int) → Bool
associativity m f g =
  ((m >>= f) >>= g) == (m >>= (λx → f x >>= g))
```

Do not forget to give the type signature as this is required by the *QuickCheck* system. To check the property, we call *quickCheck associativity* from ghci.

3 Laziness (25 POINTS)

Consider a balancing device for comparing two sets of weights. A weight will be represented by an *Int* value, and is always greater than zero. An arrangement is a pair of lists of weights: the first component of this pair corresponds to the weights on the left side of the balancing device.

$$
\begin{array}{ll}
\textbf{type } \mathit{Weight} & = \mathit{Int} \\
\textbf{type } \mathit{Weights} & = [\,\mathit{Weight}] \\
\textbf{type } \mathit{Arrangement} & = (\mathit{Weights}, \mathit{Weights})
\end{array}
$$

Given these types, we can define the following helper-functions.

$$
\begin{array}{l}
\mathit{balance} :: \mathit{Arrangement} \rightarrow \mathit{Int} \\
\mathit{balance}\ (\mathit{left}, \mathit{right}) = \mathit{sum}\ \mathit{right} - \mathit{sum}\ \mathit{left}
\end{array}
$$

$$
\begin{array}{l}
\mathit{nrOfWeights} :: \mathit{Arrangement} \rightarrow \mathit{Int} \\
\mathit{nrOfWeights}\ (\mathit{left}, \mathit{right}) = \mathit{length}\ \mathit{left} + \mathit{length}\ \mathit{right}
\end{array}
$$

The function *balance* can be used to compare the sum of the weights on both sides, whereas *nrOfWeights* counts the total number of weights of an arrangement.

The following problem has to be solved: given some weight w, and a collection of available weights ws, find an arrangement $(\mathit{left}, \mathit{right})$ such that:

- $\mathit{balance}\ (\mathit{left}, \mathit{right})$ returns w
- All available weights in ws are either used (on the left *or* on the right side) or not used. Each weight can be used only once (no duplication).
- The arrangement has the least number of weights (i.e., minimize $\mathit{nrOfWeights}\ (\mathit{left}, \mathit{right})$).

The number of arrangements to consider for a set of available weights ws is 3^n (where $n = \mathit{length}\ ws$), which quickly becomes problematic. However, we are *only* interested in finding a single arrangement with the least number of weights. Your task is to solve the problem such that arrangements using $i + 1$ weights are only considered when all possible arrangements using i weights failed. If there is no solution, then it is all right that your function inspects all 3^n cases. Rely on lazy evaluation to complete this task.

a) Write a function

$$\mathit{arrange} :: \mathit{Weight} \rightarrow \mathit{Weights} \rightarrow \mathit{Maybe}\ \mathit{Arrangement}$$

which returns a minimal arrangement for a given weight w, and a collection of available weights ws. If no arrangement exists, *Nothing* should be returned.

Example: $\mathit{arrange}\ 73\ [1, 2, 5, 10, 20, 50, 100]$ returns $\mathit{Just}\ ([2, 5, 20], [100])$. Note that two more valid arrangements exist with 4 weights.

For each weight we have three options: put it on the left-hand side, put it on the right-hand side, or don't use it. Because we only need a minimal solution, we generate a list of lists of possible solutions in which the arrangements are grouped by the number of weights involved.

```
insertSome :: Int → Weight → Arrangement → [Arrangement]
insertSome 0 _ pair      = [pair]
insertSome n w (xs, ys) = [(replicate n w ⧺ xs, ys), (xs, replicate n w ⧺ ys)]

helper :: Weights → [[Arrangement]]
helper = foldr combine start
  where
    start = [([], [])] : repeat []
    combine w xss =
      let withWeight = map (concatMap (insertSome 1 w)) xss
      in zipWith (⧺) xss ([] : withWeight)
```

With this helper function, we can give a point-free definition for *arrange*:

```
arrange i = safeHead
          ∘ filter ((== i) ∘ balance)
          ∘ concat
          ∘ takeWhile (not ∘ null)
          ∘ helper

safeHead :: [a] → Maybe a
safeHead (x : _) = Just x
safeHead _       = Nothing
```

b) Suppose we have an infinite supply of the weights that are available (the second argument): a weight can be used more than once. If we allow repeated weights, the number of arrangements to consider becomes infinite, and it is no longer straightforward to determine that no solution exists. Hence, we no longer return a *Maybe* value. Write a function

```
arrangeMultiple :: Weight → Weights → Arrangement
```

to solve the modified problem. You may re-use code fragments from the previous question.

Example: *arrangeMultiple* 73 [1, 20, 100] returns ([1, 1, 1, 1, 1, 1, 1, 20], [100]).

With duplicate weights, we still have three options for each weight: put some copies of the weight on the left-hand side, on the right-hand side, or don't use the weight at all.

```
helperMultiple :: Weights → [[Arrangement]]
helperMultiple = foldr combine start
  where
    start = [([], [])] : repeat []
    combine w xss =
      let merge (a : as) = [] : zipWith (⧺) a (merge as)
      in merge [map (concatMap (insertSome n w)) xss | n ← [0 ..]]

arrangeMultiple i = head
                  ∘ filter ((== i) ∘ balance)
                  ∘ concat
                  ∘ helperMultiple
```

4 Operational Semantics (10 POINTS)

What is the result of evaluating the following expressions (with `ghci`)? In case an expression cannot be reduced to a value, explain as precisely as possible why not.

a) $(\lambda f\ {\sim}[x] \rightarrow f\ x\ x)\ (+)\ [1\,..\,10]$

$\perp$ (pattern match failure). Even though the pattern match against the irrefutable pattern is delayed, it is performed as soon as x is needed.

b) $\mathit{null}\ \$\ (\lambda\ {\sim}(x : \mathit{xs}) \rightarrow x : \mathit{xs})\ [\,]$

False. Without doing the pattern match (that would fail) we know that the lambda expression returns a non-empty list.

c) **let** $\mathit{xs} = \mathit{replicate}\ 5 \perp$ **in** $\mathit{length}\ \$!\ \mathit{xs} +\!\!+\ \mathit{xs}$

10. The operator for strict application ($!) does not force the elements of the list to be evaluated to weak head normal form (WHNF).

d) $\mathit{seq}\ (\mathit{length}\ (\mathit{repeat} \perp))\ \mathit{True}$

$\perp$ (infinite computation). The subexpression $\mathit{length}\ (\mathit{repeat} \perp)$ cannot be reduced to a value.

e) $\mathit{length}\ [1\,..] > 10$

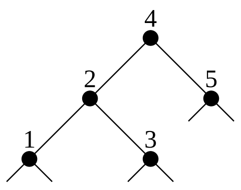
$\perp$ (infinite computation). The subexpression $\mathit{length}\ [1\,..]$ cannot be reduced to a value.

5 Higher-order Functions (15 POINTS)

Take a look at the data type *Tree*, and consider the example tree.

```
data Tree a = Bin (Tree a) a (Tree a) | Leaf deriving Show

tree :: Tree Int
tree = Bin (Bin (Bin Leaf 1 Leaf) 2 (Bin Leaf 3 Leaf)) 4 (Bin Leaf 5 Leaf)
```



a) Write a higher-order function *foldTree*, which can be used to compute a value for a tree. Also write down the type of *foldTree*. All functions in the remainder of this question have to be written with this higher-order function.

```
foldTree :: (result → a → result → result, result) → Tree a → result
foldTree alg@(bin, leaf) tree =
  case tree of
    Bin l a r → bin (foldTree alg l) a (foldTree alg r)
    Leaf      → leaf
```

b) Use *foldTree* to define the following two functions:

$$\begin{aligned} &\mathit{height} &&:: \mathit{Tree}\ a \rightarrow \mathit{Int} \\ &\mathit{mapTree} &&:: (a \rightarrow b) \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ b \end{aligned}$$

The function *height* yields the height of a tree, whereas *mapTree* applies a given function to all the elements of a tree. For example, *height tree* gives 3, and *mapTree even tree* returns *Bin* (*Bin* (*Bin Leaf False Leaf*) *True* (*Bin Leaf False Leaf*)) *True* (*Bin Leaf False Leaf*).

$$\begin{aligned} &\mathit{height} &&= \mathit{foldTree}\ (\lambda l\ \_\ r \rightarrow 1 + \mathit{max}\ l\ r, 0) \\ &\mathit{mapTree}\ f &&= \mathit{foldTree}\ (\lambda l\ a\ r \rightarrow \mathit{Bin}\ l\ (f\ a)\ r, \mathit{Leaf}) \end{aligned}$$

c) Use *foldTree* to write a function for collecting the elements of a tree in a depth-first order.

$$\mathit{depthfirst} :: \mathit{Tree}\ a \rightarrow [a]$$

For instance, *depthfirst tree* gives $[1,3,2,5,4]$. Take into account that concatenation of two lists $(+\!\!+)$ requires a traversal over the left operand: prevent quadratic behavior for *depthfirst*.

$$\begin{aligned} &\mathit{depthfirst}\ \mathit{tree} = \\ &\quad \mathit{foldTree}\ (\lambda f\ a\ g \rightarrow f \circ g \circ (a{:}), \mathit{id})\ \mathit{tree}\ [\,] \end{aligned}$$

d) Use *foldTree* to write a function for collecting the elements of a tree in a breadth-first order.

$$\mathit{breadthfirst} :: \mathit{Tree}\ a \rightarrow [a]$$

For instance, *breadthfirst tree* gives $[4,2,5,1,3]$. For this part, you don't have to worry about the efficiency of $(+\!\!+)$.

Idea: use a list of lists. Elements in the same list appear at the same depth in the tree.

$$\begin{aligned} \mathit{breadthfirst} = {} & \mathit{concat} \\ & \circ\ \mathit{takeWhile}\ (\mathit{not} \circ \mathit{null}) \\ & \circ\ \mathit{foldTree}\ (\lambda t1\ a\ t2 \rightarrow [a] : \mathit{zipWith}\ (+\!\!+)\ t1\ t2, \mathit{repeat}\ [\,]) \end{aligned}$$