Department of Information and Computing Sciences
Utrecht University

INFOAFP – Exam

Andres Löh

Monday, 19 April 2010, 09:00–12:00

Preliminaries

- The exam consists of 7 pages (including this page). Please verify that you got all the pages.
- A maximum of 100 points can be gained.
- For every task, the maximal number of points is stated. Note that the points are distributed unevenly over the tasks.
- One task is marked as (*bonus*) and allows up to 5 extra points.
- Try to give simple and concise answers! Please try to keep your code readable!
- When writing Haskell code, you can use library functions, but make sure that you state which libraries you use.

Good luck!

Zippers (33 points total)

A *zipper* is a data structure that allows navigation in another tree-like structure. Consider binary trees:

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving (Eq, Show)
```

A *one-hole context* for trees is given by the following datatype:

```haskell
data TreeCtx a = NodeL () (Tree a) | NodeR (Tree a) ()
  deriving (Eq, Show)
```

The idea is as follows: leaves contain no subtrees, therefore they do not occur in the context type. In a node, we can focus on either the left or the right subtree. The context then consists of the other subtree. The use of () is just to mark the position of the hole – it is not really needed.

We can plug a tree into the hole of a context as follows:

```haskell
plugTree :: Tree a → TreeCtx a → Tree a
plugTree l (NodeL () r) = Node l r
plugTree r (NodeR l ()) = Node l r
```

A zipper for trees encodes a tree where a certain subtree is currently in focus. Since the focused tree can be located deep in the full tree, one element of type *TreeCtx a* is not sufficient. Instead, we store the focused subtree together with a *list* of one-layer contexts that encodes the path from the focus to the root node:

```haskell
data TreeZipper a = TZ (Tree a) [TreeCtx a]
  deriving (Eq, Show)
```

We can recover the full tree from the zipper as follows:

```haskell
leave :: TreeZipper a → Tree a
leave (TZ t cs) = foldl plugTree t cs
```

Consider the tree

```haskell
tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b'))
            (Node (Leaf 'c') (Leaf 'd'))
```

If we focus on the rightmost leaf containing 'd', the corresponding zipper structure is

```haskell
example :: TreeZipper Char
example = TZ (Leaf 'd')
             [NodeR (Leaf 'c') (), NodeR (Node (Leaf 'a') (Leaf 'b')) ()]
```


2

1 (3 points)**.** Define a function

```
enter :: Tree a → TreeZipper a
```

that creates a zipper from a tree such that the full tree is in focus. •

Moving the focus from a tree down to the left subtree works as follows:

```
down :: TreeZipper a → Maybe (TreeZipper a)
down (TZ (Leaf x)   cs) = Nothing
down (TZ (Node l r) cs) = Just (TZ l (NodeL () r : cs))
```

The function fails if there is no left subtree, i. e., if we are in a leaf.

2 (8 points)**.** Define functions

```
up    :: TreeZipper a → Maybe (TreeZipper a)
right :: TreeZipper a → Maybe (TreeZipper a)
```

that move the focus from a subtree to its parent node or to its right sibling, respectively. Both functions should fail (by returning *Nothing*) if the move is not possible. •

3 (6 points)**.** Assuming a suitable instance

```
instance Arbitrary a ⇒ Arbitrary (TreeZipper a)
```

consider the QuickCheck property

```
downUp :: (Eq a) ⇒ TreeZipper a → Bool
downUp z = (down z ≫= up) == Just z
```

Give a counterexample for this property, and suggest how the property can be improved so that the test will pass. •

4 (4 points)**.** Is

```
left :: TreeZipper a → Maybe (TreeZipper a)
left z = up z ≫= down
```

a suitable definition for *left*? Give reasons for your answer. [No more than 30 words.]
•

5 (6 points)**.** The concept of a *one-hole context* is not limited to binary trees. Give a suitable definition of *ListCtx* such that we can define

```
data ListZipper a = LZ [a] [ListCtx a]
```

and in principle play the same game as with the zipper for trees. Also define the function

```
plugList :: [a] → ListCtx a → [a]
```

the combines a list context with a list. •

6 (6 points)**.** Discuss the necessity of *up*, *down*, *left* and *right* functions for the *ListZipper*, and describe what they would do. No need to define them (although it is ok to do so). [No more than 40 words.] •

Type isomorphisms (12 points total)

7 (6 points)**.** A different definition for one-hole contexts of trees is the following:

$$\mathbf{data}\ \mathit{Dir} = L \mid R$$
$$\mathbf{type}\ \mathit{TreeCtx}'\ a = (\mathit{Dir}, \mathit{Tree}\ a)$$

Show that, ignoring undefined values, the types *TreeCtx* and *TreeCtx′* are isomorphic, by giving conversion functions and stating the properties that the conversion functions must adhere to (*no proofs required*). •

8 (6 points)**.** In Haskell's lazy setting, how many different values are there of type *TreeCtx Bool* if we restrict the occurrences of *Tree Bool* to be leaves. And how many different values are there of type *TreeCtx′ Bool* given the same restriction? (Hint: note that the use of () in the definition of *TreeCtx* is relevant here.) •

Lenses (14 points total, plus 5 bonus points)

A so-called *lens* is (among other things) a way to access a substructure of a larger structure by grouping a function to extract the substructure with a function to update the substructure:

$$\begin{aligned} \mathbf{data}\ a \mapsto b = \mathit{Lens}\ \{ & \mathit{extract} :: a \to b, \\ & \mathit{insert}\ \ :: b \to a \to a \} \end{aligned}$$

(We assume here that we enable infix type constructors, and that $\mapsto$ is a valid symbol for such a constructor.)

Lenses are supposed to adhere to the following two *extract*/*insert* laws:

$$\begin{aligned} &\forall (f :: a \mapsto b)\ (x :: a). && \mathit{insert}\ f\ (\mathit{extract}\ f\ x)\ x \equiv x \\ &\forall (f :: a \mapsto b)\ (x :: b)\ (y :: a). && \mathit{extract}\ f\ (\mathit{insert}\ f\ x\ y) \equiv x \end{aligned}$$

A trivial lens is the identity lens that returns the complete structure:

$$\mathit{idLens} :: a \mapsto a$$
$$\mathit{idLens} = \mathit{Lens}\ \{\mathit{extract} = \mathit{id}, \mathit{insert} = \mathit{const}\}$$

It is trivial to see that *idLens* fulfills the two laws.

9 (4 points)**.** Define a lens that accesses the focus component of a tree zipper structure:

$$\mathit{focus} :: \mathit{TreeZipper}\ a \mapsto \mathit{Tree}\ a$$

•

10 (4 points)**.** Define a function that updates the substructure accessed by a lens according to the given function:

$$\mathit{update} :: (a \mapsto b) \to (b \to b) \to (a \to a)$$

•

Lenses can be composed. Structures that support identity and composition are captured by the following type class:

```
class Category cat where
  id  :: cat a a
  (∘) :: cat b c → cat a b → cat a c
```

For instance, functions are an instance of the category class, with the usual definitions of identity and function composition:

```
instance Category (→) where
  id  = Prelude.id
  (∘) = (Prelude.∘)
```

11 (6 points)**.** Define an instance of the *Category* class for lenses:

```
instance Category (↦) where
  ...
```

•

12 (5 *bonus points*)**.** Prove using equational reasoning that if the two *extract*/*insert* laws stated above hold for both f and g, then they also hold for $f \circ g$. •

Monad transformers (22 points total)

Consider the monad *TraverseTree*, defined as follows:

```
type TraverseTree a = StateT (TreeZipper a) Maybe
```

13 (3 points)**.** What is the kind of *TraverseTree*? •

14 (6 points)**.** Define a function

```
nav :: (TreeZipper a → Maybe (TreeZipper a)) → TraverseTree a ()
```

that turns a navigation function like *down*, *up*, or *right* into a monadic operation on *TraverseTree*. •

Given a lense and the *MonadState* interface, we can define useful helpers to access parts of the monadic state:

```
getLens :: MonadState s m ⇒ (s ↦ a) → m a
getLens f = gets (extract f)
putLens :: MonadState s m ⇒ (s ↦ a) → a → m ()
putLens f x = modify (insert f x)
```

```
modifyLens :: MonadState s m ⇒ (s ↦ a) → (a → a) → m ()
modifyLens f g = modify (update f g)
```

We can now define the following piece of code:

```
ops :: TraverseTree Char ()
ops =
  do
    nav down
    x ← getLens focus
    nav right
    putLens focus x
    nav down
    modifyLens focus (const $ Leaf 'X')
```

15 (6 points)**.** Given all the functions so far and once again tree

```
tree = Node (Node (Leaf 'a') (Leaf 'b'))
            (Node (Leaf 'c') (Leaf 'd'))
```

what is the result of evaluating the following declaration:

```
test = leave (snd (fromJust (runStateT ops (enter tree))))
```

•

16 (7 points)**.** Explain how a compiler based on passing dictionaries for type classes can construct the dictionary to pass to the *modifyLens* call in the last line of the definition of *ops* above. •

Trees, shapes and pointers in Agda (19 points total)

Consider the definitions of *List*, **N**, *Vec* and *Fin* in Agda. These four types are related as follows:

Natural numbers describe the *shapes* of lists (if we instantiate the element type of lists to the unit type, we obtain a type isomorphic to the natural numbers). Indexing lists by their shapes yields vectors. Finally, *Fin* is the type of *pointers* into vectors such that we can define a safe lookup function.

Now consider binary trees (as before), given in Agda by:

```
data Tree (A : Set) : Set where
  leaf  : A → Tree A
  node : Tree A → Tree A → Tree A
```

The type of shapes for trees is given by:

```
data Shape : Set where
  end   : Shape
  split : Shape → Shape → Shape
```

17 (5 points). Define a datatype *STree* of shape-indexed binary trees (i. e., *STree* corresponds to *Vec*):

$$\mathbf{data}\ \mathit{STree}\ (A : \mathit{Set}) : \mathit{Shape} \to \mathit{Set}\ \mathbf{where}$$
$$\ldots$$

18 (6 points). Define a datatype *Path* of shape-indexed pointers (i. e., *Path* corresponds to *Fin*):

$$\mathbf{data}\ \mathit{Path} : \mathit{Shape} \to \mathit{Set}\ \mathbf{where}$$
$$\ldots$$

Note that a value p of type $\mathit{Path}\ s$ should point to an element in a tree of shape s.

19 (4 points). Define a function *zipWith* on shape-indexed trees that merges two trees of the same shape and combines the elements according to the given function.

$$\begin{aligned}\mathit{zipWith} : {} & \forall \{A\ B\ C\ s\} \to (A \to B \to C) \to \\ & \mathit{STree}\ A\ s \to \mathit{STree}\ B\ s \to \mathit{STree}\ C\ s\end{aligned}$$

20 (4 points). Define a function *lookup* on shape-indexed trees

$$\mathit{lookup} : \forall \{A\ s\} \to \mathit{STree}\ A\ s \to \mathit{Path}\ s \to A$$

that returns the element stored at the given path.