Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming

WS 2019/20
Benjamin Dietrich, Denis Hirn

Assignment #9

Submission Deadline: Thu, 16.1.2020

Exercise 1: Monoids (5 Points)

Let's talk about *monoids* and trees, more specifically the following type of rose trees with labels of type a:

```
data Tree a = Node a [Tree a]
```

Please hand in a file `Monoid.hs`. Import the module `Data.Monoid` and have a look at the documentation for that module[1].

1. Write a function `sumTree :: Num a => Tree a -> a` that computes the *sum* of all node labels in a tree.
2. Write a function `treeLabels :: Tree a -> [a]` that computes the *list* of all node labels in a tree.
3. We continue our hunt for common patterns in computations that we might abstract over. `sumTree` and `treeLabels` are suspiciously similar: The label of the current node is combined with the results for all subtrees. We abstract over this pattern in a function

   ```
   foldTree :: Monoid m => (a -> m) -> Tree a -> m
   ```

   Given a function that maps a node label to some element of monoid `m`, `foldTree` combines the monoidal result for the node label and the results for all subtrees.

   Implement `foldTree`.
4. First, use `foldTree` to implement a function `treeLabels' :: Tree a -> [a]` that behaves like `treeLabels`.

   Then, do the same for `sumTree`. Remember from the lecture that there is no single `Monoid` instance for numeric types. Instead, we have `newtype` wrappers `Sum` and `Product` whose `Monoid` instances implement the additive and multiplicative monoids for numeric type a, respectively
5. Finally, implement functions

   ```
   allNodes :: (a -> Bool) -> Tree a -> Bool
   someNode :: (a -> Bool) -> Tree a -> Bool
   ```

   that check whether *all* or *some* node labels in a tree satisfy a predicate.

   Again, we do not have a single `Monoid` instance for `Bool`, but wrappers `Any` and `All` that implement monoids with different behaviours.

[1] http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Monoid.html

Exercise 2: Applicative Zip-Lists (5 Points)

In the lectures we discussed the `Applicative` instance for lists as implemented in `Prelude`; for `<*>` each function of the left argument list is applied to *each* value of the right argument list:

```
Prelude> [(*2),(+2)] <*> [21,40,7]
[42,80,14,23,42,9]
```

However, this is not the only possible way to define a useful instance of `Applicative` for lists. As with `Product` and `Sum` for alternative instances of `Monoid`, we can define a `newtype Zip a` to implement an alternative instance of `Applicative` for lists.

Consider the following behavior of `<*>` for these `Zip`-lists:

```
Prelude> Zip [(*2),(+2)] <*> Zip [21,40,7]
Zip [42,42]
```

On the level of values, each function of the left argument list is applied to the *one* value that is on the same position of the right argument list. On the level of structures, the lists are combined to a list with the length of the shorter input list.

1. Define a `newtype Zip a` for lists of values of type `a`.
2. Make `Zip a` an instance of `Applicative`. As for now, assume `pure x = Zip [x]` and define the *tie-fighter* operator `<*>` as described above.

The implementation of `pure` suggested above violates a law, that is not ensured by the Haskell compiler, but all `Applicative` instances must satisfy. The *identity* rule documented for the `Applicative` class[2] requires for all possible inputs `v` that:

`pure id <*> v` $\equiv$ `v`

In other words: A computation which does neither touch the structure (`pure`) nor affect the inner value (`id`) must not have any effect at all.

3. Give an example which shows that the *identity* rule is violated.
4. Implement a definition of `pure` that does not violate the *identity* rule for any possible Zip-list `v`.

The `Zip` applicative instance can now be used in alternative to all `zip`N and `zipWith`N functions in `Data.List`.

5. Reformulate the following expression to use only `<$>` and `<*>` instead of `zipWith3`:

```
zipWith3 (\a b c -> a + b * c) [1,2,3] [4,5,6] [7,8]
```

[2]https://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#t:Applicative

Exercise 3: Edit Distance **(10 Points)**

Think of a constantly changing text whose current version is to be printed on a rather slow live-display. To save costs in communication and transformation, changes in the text can be expressed by a minimal *sequence of modification operations*. Your task is to find a modifying sequence which transforms a given text t_1 into another given text t_2 with minimal *cost*.

The supported modification operations are given as follows (all in terms of the *current position* in text t_1 which is traversed from left to right):

- `Change` c: Replace the character at the current position with a new character c.
- `Insert` c: Insert a new character c before the current position.
- `Copy`: Copy the character at the current position without changes.
- `Delete`: Delete the character at the current position.
- `Kill`: Delete the rest of the text, starting from the current position.

For instance, take the transformation from $t_1 =$ `"fish"` to $t_2 =$ `"chips"` (current position underlined):

$$\texttt{\underline{f}ish} \xrightarrow{\texttt{Insert 'c'}} \texttt{c\underline{f}ish} \xrightarrow{\texttt{Change 'h'}} \texttt{ch\underline{i}sh} \xrightarrow{\texttt{Copy}} \texttt{chi\underline{s}h} \xrightarrow{\texttt{Insert 'p'}} \texttt{chip\underline{s}h} \xrightarrow{\texttt{Copy}} \texttt{chips\underline{h}} \xrightarrow{\texttt{Kill}} \texttt{chips}$$

1. Define a data type `Edit` to represent the modification operations described above.
2. Write a function `transform :: String -> String -> [Edit]` which determines the sequence of modification operations transforming a given text t_1 to another text t_2 with the lowest cost. Assume that each operation of `Edit` has costs of exactly *one* unit, except for `Copy` which is performed for free.
   Hint: Walk throw t_1 and t_2 in parallel from left to right. For each position, try all modification operations and choose the cheapest one.
3. Write a function `traceTransform :: String -> [Edit] -> String` that generates a log of the sequence of modifications applied to the given string.

   Example:

   ```
   > putStr $ traceTransform "time" $ transform "time" "flies"

                  time
   Insert 'f'     ftime
   Change 'l'     flime
   Copy           flime
   Delete         flie
   Copy           flie
   Insert 's'     flies
   ```