Prof. Dr. Peter Thiemann
Janek Spaderna
janek.spaderna@pluto.uni-freiburg.de

Winter Term 2022/23
Issued 2023-02-05
Discussion on 2023-02-10

Functional Programming
`https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/`

Exercise Sheet 13

Download `ALaCarte.hs` linked on the lecture page. It contains most of the definitions from the two *Datatypes à la carte* lectures.

Exercise 1 (Arbitrary à la carte)

Write `Arbitrary` instances for the types `(f :+: g) a` and `Mu f` (of course restricted by suitable constraints) and the expression functors (`Val e`, `Add e`, ...). The naïve implementation for `(:+:)` is heavily biased towards the left. For example, for a type like $\texttt{F}_1$ `:+:` ($\texttt{F}_2$ `:+:` ($\texttt{F}_3$ `:+:` $\texttt{F}_4$)) `()` it produces (on average) twice as many values from $\texttt{F}_1$ as from $\texttt{F}_4$. Your implementation should do better.

Hint This involves writing and implementing a custom typeclass to, for example, collect all potential generators before applying a combinator such as `oneof`.

Bonus In an unfortunate case of bad luck the generator could return infinitely large trees. Put safeguards in place to limit the maximum recursion depth. QuickCheck provides `getSize`, `resize`, and similar functions. How the user makes use of the size parameter is mostly up to them. A greater size, however, should in general lead to a larger result.

Exercise 2 (Partial rewriting)

Using the technique from *Datatypes à la carte* our datatypes are built from small building blocks. This representation lends itself for a step-by-step rewriting of the tree. For example, think of a data structure which is refined during the program's execution with additional data; or some constructors are replaced by combinations of others to reduce the overall amount of cases to consider in later stages of the program.

1. Define functors to represent a simple functional language consisting of
   - variables
   - function application
   - lambda abstraction
   - let bindings
2. We consider let-bindings as syntax sugar, which we don't want to handle at every point in our program. Luckily, an expression `let` x `=` $\texttt{e}_1$ `in` $\texttt{e}_2$ can be rewritten as a combination of application and abstraction: $(\lambda x.\ \texttt{e}_2)\ \texttt{e}_1$.

   Implement the function `rewriteLets :: ... => Mu (Let :+: f) -> Mu f`, which performs this rewrite step. (Where `Let` is the functor representing let-bindings.)

For the next step we need some additional background. There are two typeclasses in the base library which haven't made an appearance yet: `Foldable` and `Traversable`. They build upon `Functor` orthogonally to `Applicative` and `Monad`.

A `Foldable` instance is characterised by its `foldMap` function. For a parameterized datatype `F a` it combines every occurrence of `a` into one final value—or as the documentation puts it

> The `Foldable` class represents data structures that can be reduced to a summary value one element at a time.

The **`Traversable`** class is explained as follows:

> **`Traversable`** structures support element-wise sequencing of **`Applicative`** effects (thus also **`Monad`** effects) to construct new structures of *the same shape* as the input.
>
> To illustrate what is meant by *same shape,* if the input structure is `[a]`, each output structure is a list `[b]` of the same length as the input. If the input is a `Tree a`, each output `Tree b` has the same graph of intermediate nodes and leaves. Similarly, if the input is a 2-tuple `(x, a)`, each output is a 2-tuple `(x, b)`, and so forth.
>
> It is in fact possible to decompose a traversable structure `t a` into its shape (a. k. a. *spine*) of type `t ()` and its element list `[a]`. The original structure can be faithfully reconstructed from its spine and element list.
>
> *From the Overview section in `Data.Traversable`. Read more there.*

3. Define **`Foldable`** and **`Traversable`** instances for the `(:+:)` combinator and the syntax functors.

   Hint If you have trouble understanding the involved type signatures or can't make sense of the operations look at the concrete cases and let yourself be guided by the types.

4. Using these prerequisits write a function to perform the annotation of abstraction nodes described above. That is, replace abstraction nodes with new nodes containing an additional piece of information. It should indicate whether the variable bound by this abstraction is used inside its body.

Exercise 3 (Free monads cut off)

Using free monads we have access to the sequence of monadic actions and can manipulate them before we begin evaluating. In this exercise you will write a function **`cutoff`**. Given an integer n it will transform a **`Term f a`** to evaluate to **`Nothing`** if the computation does not complete after interpreting at most n operations. If the computation does complete, the result should be returned wrapped in **`Just`**.

Implement `cutoff :: Functor f => Integer -> Term f (Maybe a)`.

Some properties it should satisfy:

```
prop_cutoff0    n t = n <= 0 ==> cutoff n t == return Nothing
prop_cutoffPure n a = n  > 0 ==> cutoff n (return a) == return (Just a)
```