Monad-Based Programming SS 2023

Assignment 3

Deadline for solutions: 08.06.2023

Exercise 1 Traversing Trees (3 Points)

The following code implements a breadth-first traversal of a tree, using the standard implementation of trees from Data.Tree:

```
import Data.List
import Data.Monoid
import Data.Tree (Tree(..))

newtype BFSTree a = BFS (Tree a)

instance Foldable (BFSTree) where
  foldMap f (BFS tr) = go [tr]
    where
      go q = case q of
        [] -> mempty
        (Node x xs) : qs -> f x `mappend` go (qs ++ xs)
```

Hence one can run programs like

```
foldr (:) [] (BFS some_tree)
```

to obtain a breadth-first unfolding of a tree some_tree into a list, instead of the default depth-first unfolding by

```
foldr (:) [] some_tree
```

In contrast to the depth-first strategy, the breadth-first search strategy fully traverses even a tree with infinitely many nodes.

```
t = Node "1(0)" [Node "2(1)" [Node "4(2)" [], Node "5(2)" []], Node "3(1)" []]
```

Then

```
foldr (:) [] t        == ["1(0)","2(1)","4(2)","5(2)","3(1)"]
foldr (:) [] (BFS t)  == ["1(0)","2(1)","3(1)","4(2)","5(2)"]
```

(a) This implementation is based on using the list type [BFSTree] as a *queue* in which new trees are added at the back with the qs ++ xs command, which is highly inefficient, because it requires full traversal of the pending queue qs at every iteration. Reimplement foldMap, so that it behaves the same, but does not suffer from this issue. Compare the performance of both implementations by running tests on exponentially growing trees, e.g.

```
expTree a b = Node (a, b) [expTree (a + 1) b, expTree a (b + 1)]
```

(b) Analogously, implement a strategy that zig-zags through the three in a breadth-first fashion, e.g.:

```
foldr (:) [] (ZZS t) == ["1(0)","2(1)","3(1)","5(2)","4(2)"]
```

Exercise 2 PCF in Haskell (4 Points)

PCF-Terms can be internally represented in Haskell in the following way.

```haskell
-- Alias type for variable names as strings
type Name = String

-- Language Terms
data Expr = T | F
  | IfThenElse Expr Expr Expr
  | Nmb Int
  | Sum Expr Expr
  | Dif Expr Expr
  | Eq Expr Expr
  | Var Name
  | App Expr Expr
  | Lam Name Expr
  | Fix
    deriving (Eq, Show)
```

For example, we can define the following familiar terms:

```haskell
omega = App Fix (Lam "x" (Var "x"))
church0 = Lam "z" (Lam "n" (Var "z"))
church1 = Lam "z" (Lam "n" (App (Var "n") (Var "z")))
church2 = Lam "z" (Lam "n" (App (Var "n") (App (Var "n") (Var "z"))))
```

(of course, nothing prevents one from forming nonsensical terms, which do not correspond to typable PCF-terms, like `App (Nmb 0) (Nmb 0)`; there is a technique to faithfully represent typed languages within Haskell [1], but we do not have capacities to dwell on it).

Implement call-by-name small-step reduction as a Haskell function:

```haskell
eval :: Expr -> Expr
```

so, that `t' = eval t` iff $t \to_{cbn} t'$ for all correctly typed terms t, t' (implementing type checking is not required).

Exercise 3 Lazy Lists (8 Points)

(a) Complete the PCF language, described at the lecture with

- type constructors for forming coproduct types $A+B$ (corresponding to Haskell's **Either** A B) and for forming lists $A^\star$ (corresponding to Haskell's [A]),
- with the corresponding constructors inl, inr (corresponding to **Left** and **Right**), nil and cons (corresponding to [ ] and (:)),
- and with the corresponding case-statements that pattern-match on the new constructors (take inspiration from PCF's if-then-else and Haskell's case operator).

More precisely, you need to: add the new rules for forming terms-in-contexts, to modify the definition of value and to add the new (big-step or small-step) call-by-name operational semantics rules.

Hint: Do the case of coproducts first and then switch to lists. In both cases, success depends on understanding the relevant notion of value correctly; recall that values are irreducible closed terms, w.r.t. the evaluations strategy of interest — it is thus advisable to fully describe what a value is in each case and then produce the corresponding rules of the operational semantics.

(b) Extend the interpreter from Exercise 1 accordingly.

(c) How the notion of value would change if we considered the call-by-value semantics?

(d) Translate the following Haskell program

```
fib = 1 : 1 : [a + b | (a, b) <- zip fib (tail fib) ]
```

to the above described extended PCF. **Hint:** For a start, desugar the above Haskell program suitably, to make the syntax closer to that of PCF, e.g. `[f a b | (a, b) <- xs]` can be replaced with `[f (fst x) (snd x) | x <- xs]`, and the latter can be modelled by the **map** function. You then need to implement the relevant auxiliary functions (**zip**, **map**, ...) in PCF and define `fib` using them.

(e) Analogously, translate the program `(fib !! 4)` to a PCF program p and prove that $p \Downarrow 5$, alternatively that $p \to^\star 5$ in the small-step semantics style. It is also OK to submit a program that generates the requisite transition (chain), e.g. by running the interpreter from part (b).

Exercise 4 `curry` and `uncurry` (5 Points)

Given two complete partial orders A and B, let $A \to_c B$ be the space of continuous functions from A to B, more precisely

$$A \to_c B = \{f \colon A \to B \mid A \to B, \text{ such that } f \text{ is continuous}\}.$$

Note that $A \to_c B$ is a partial order under the pointwise extension from B, i.e. $f \sqsubseteq g$ for $f, g \colon A \to_c B$ if $f(x) \sqsubseteq g(x)$ for all $x \in A$.

(a) Show that $A \to_c B$ are complete partial orders, if A and B are so;

(b) Given complete partial order A, B and C, show that $\mathtt{curry} \colon (C \times B \to_c A) \to (C \to_c (A \to_c B))$ and $\mathtt{uncurry} \colon (C \to_c (A \to_c B)) \to (C \times B \to_c A)$ are monotone and continuous where

$$(\mathtt{curry}\ f)(x)(y) = f(x, y)$$
$$(\mathtt{uncurry}\ f)(x, y) = f(x)(y)$$

(you are encouraged to experiment with `ghci`, in which these functions are available under the same name). **Hint:** Use the fact from the lecture that $A \times B$ is a complete partial order whenever A and B are, where $(a, b) \sqsubseteq (a', b')$ means by definition that $a \sqsubseteq a'$ in A. and $b \sqsubseteq b'$ in B;

References

[1] Danvy, Olivier, and Morten Rhiger. 'A Simple Take on Typed Abstract Syntax in Haskell-like Languages'. BRICS Report Series, no. 34, June 2000. https://doi.org/10.7146/brics.v7i34.20169.