Selected Programming Techniques

This chapter focuses on functional programming techniques, often used to make functional programs more efficient.

Difference listes

A well known example of program optimization for a program on lists is the `reverse`-function. The naive implementation with quadratic run time is:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Using an accumulator, it is possible to realize a linear algorithm:

```
reverse :: [a] -> [a]
reverse l = rev l []
 where
  rev :: [a] -> [a] -> [a]
  rev []     ys = ys
  rev (x:xs) ys = rev xs (x:ys)
```

The second implementation is less intuitive, but much more efficient. The first implementation is based on the induction principle, without introducing an additional parameter, which is used like a kind of stack and reverses the list.

Because of currying, we can see `rev` also as a function of arity one, which yields a function of type `[a] -> [a]`. Its type is 'rev :: [a] -> ([a] -> [a]). Using this idea, we can also implement rev as follows:

```
reverse :: [a] -> [a]
reverse l = rev l []
 where
  rev []     = id
  rev (x:xs) = rev xs . (x:)
```

Now, `rev` has the same structure as the naive implementation of reverse. `id` is the value representing the empty list and function composition `(.)` takes the role of `(++)`. Finally, the one elementary list `[x]` is represented by the section `(x:)`. So, we replaced the list monoid by the function moniod and optimised efficiency of reverse.

This idea can be expressed within an abstract data type, called difference lists[1]. Another name, often used for this data structure is functional list, since the list is represented by a function.

[1]The name *difference list* comes from logic programming - an analogy we don't discuss any further here.

```haskell
import Data.Monoid

newtype DList a = DList ([a] -> [a])

instance Monoid (DList a) where
  mempty                        = DList id
  DList xs `mappend` DList ys = DList (xs . ys)
```

The implementation of `mappend` is non-recursive and hence takes constant run time, which then results in a linear algorithm for `reverse`.

Like in the implementation of `reverse` it is always possible, to convert a difference list into a common list, by simply applying the difference list to the empty list.

```haskell
fromDList :: DList a -> [a]
fromDList (DList xs) = xs []
```

Intuitively, a difference list is a function, that holds elements and is able to put this elements in front of it missing argument. A common list can therefore be converted into a difference list by

```haskell
toDList :: [a] -> DList a
toDList xs = DList (xs ++)
```

Now we can now re-implement the `reverse`-function, to make it more visible that we are using difference lists.

```haskell
reverse :: [a] -> [a]
reverse l = fromDList (rev l)
 where
  rev []     = mempty
  rev (x:xs) = rev xs `mappend` toDList [x]
```

This implementation uses the `(++)`-function only within `toDList` and only with the one elementary list `[x]`. Hence, its run-time is similar to the implementation using the accumulator technique. But it is as elegant, as the naive implementation of `reverse`.

The application of difference lists is not restricted to `reverse`. Whenever one constructs a list, e.g. while traversing a tree, difference lists are a good choice. As an example, we consider a function, collecting node labels in infix order. The naive implementation uses the result of the recursive call as left argument of `(++)` and in worst-case (unbalanced tree to the left) we obtain qudratic run-time in the number of nodes in the tree.

With difference lists we can list the labels in linear time with respect to the number of nodes inside the tree:

```haskell
data Tree a = Empty
            | Node (Tree a) a (Tree a)
 deriving Eq
```

```
infixLabels :: Tree a -> [a]
infixLabels = fromDList . labels
 where
  labels Empty          = mempty
  labels (Branch l x r) =
    labels l `mappend` toDList [x]
             `mappend` labels r
```

The functions `toDList` and `fromDList` are inverse `monoid`-isomorphisms to each other (simple proof by induction). Therefore, it is possible to translate every program using lists and only using monoid operations into an equivalent program using difference lists, without changing its behaviour.

Unfortunately, we cannot use difference list in every program operating on lists. For instance, we cannot check whether a give `DList` is empty, without converting it into the common list form.

```
nullDL :: DList a -> Bool
nullDL (DList dl) = null (dl [])
```

We have to apply the `DList` to the empty list and can perform the empty test afterwards. Otherwise, we cannot see the top-level constructor. For this problem, we can still find a solution, by storing the size of the list in an additional `Int`-parameter for `DList`, which would also allow us, to define a `length` function for `DLists`. However, this trick is restricted to simple information. We cannot access the elements within the difference list and can therefore not implement functions like `map`- or `concat` on difference lists, as the following try shows:

```
mapDL :: (a -> b) -> DList a -> DList b
mapDL f (DList dl) = DList (\l -> ???)
```

We are not able to apply `f` to the elements of `dl`, without converting `dl` into a common list. This also holds for many other list functions.

However, we will later introduce some advanced techniques, which will allow the definition of `map` and `concat` for difference lists.

Another typical example for tree traversals, which produces lists, are `show`-functions. The `Show`-class contains the following functions[2]:

```
class Show a where
  show :: a -> String
```

[2] In addition to the presented functions, the class `Show` provides a functions `showList :: Show a => [a] -> String`. By this function, it is possible to modify the list representation of a given data type. The default implementation presents lists with square brackets and separates the elements with commas. For the data type `Char`, however, we want a representation of lists of `Char` as a `String`. Therefore, in the `Char`-instance, the function `showlist` is overwritten, such that lists of characters are represented in the well known string representation.

```
  show x = shows x ""

  showsPrec :: Int -> a -> ShowS
```

The type `ShowS` and the function `shows` are defined as follows:

```
type ShowS = String -> String

shows :: Show a => a -> ShowS
shows = showsPrec 0
```

The type `ShowS` represents strings as difference lists and `shows` behaves like `show`, but with difference lists. The function `showsPrec` gets an additional parameter, which can be used to respect precedences, to avoid superflous brackets. For simplicity, we will ignore this parameter in the following.

Using the type `ShowS`, we can convert tree data structures into strings in linear time with respect to the size of the tree.

```
instance Show a => Show (Tree a) where
  showsPrec _ Empty          = showString "Empty"
  showsPrec _ (Branch l x r) = showString "Branch " .
                               showParen (l/=Empty) (shows l) .
                               showChar ' ' . shows x . showChar ' ' .
                               showParen (r/=Empty) (shows r)
```

Similar to the type `DList` we use function composition instead string concatenation.

The function `showString` relates to the function `toDList` and constructs from a `String` a corresponding value of type `ShowS`.

```
showString :: String -> ShowS
showString = (++)
```

Similar, `showChar` behaves for single characters

```
showChar :: Char -> ShowS
showChar = (:)
```

and `showParen` constructs parentheses around a value within `ShowS`, if the flag passed is `True`.

```
showParen :: Bool -> ShowS -> ShowS
showParen True  s = showChar '(' . s . showChar ')'
showParen False s = s
```

All these functions are defined in the `Prelude`.

Continuations

As an example for the monad `Maybe` we defined a data type

```
data Expr = Num Int
          | Expr :+: Expr
          | Expr :/: Expr
```

and a function

```
eval :: Expr -> Maybe Int
```

The result of `eval` was `Nothing`, if a division by zero occurred during the evaluation.

Let's consider the evaluation of the following expression:

```
Num 4 :/: (Num 1 :+: Num (-1)) :+: Num 4
```

The pass through the tree representing the data structure can be visualised by the following picture.

If the value `Nothing` occurs, we cancel the walk through the tree. For instance the right arguemtn of the top-level `:+:`-node is not evaluated, because the left arguemnt fails.

The value `Nothing` occurs somewhere in the tree and is sucessively passed to the root of the tree. The `eval`-function or more precise, the bind operator of the `Maybe`-Monad testes every result, whether it is `Nothing` and then produces `Nothing` as its result. If the first `Nothing` occures very deep in the tree, this passing of `Nothing` to the root may be a bit inefficient. Is it possible to stop the whole computation instead of passing `Nothing` values up to the root?

A solution for this is called *Continuation Passing Style (CPS)* and we define a CPS version of our `eval`-function. A function in CPS takes a function as an additional argument, the so called *continuation*, which represents the whole further computation. In CPS, the type of the `eval`-function changes to:

```
evalCPS :: Expr
        -> (Int -> Maybe Int)
        -> Maybe Int
```

Since the continuation represents the further computation, we can simply apply it in the success case. For the leafs in our expressions we get:

```
evalCPS (Num x) k = k x
```

To evaluate multiple expressions successively and combine the results, we nest the continuations and evaluate the second expression within the continuation of the first expression.

```
evalCPS (e1 :+: e2) k =
  evalCPS e1 (\v1 -> evalCPS e2 (\v2 -> k (v1 + v2)))
```

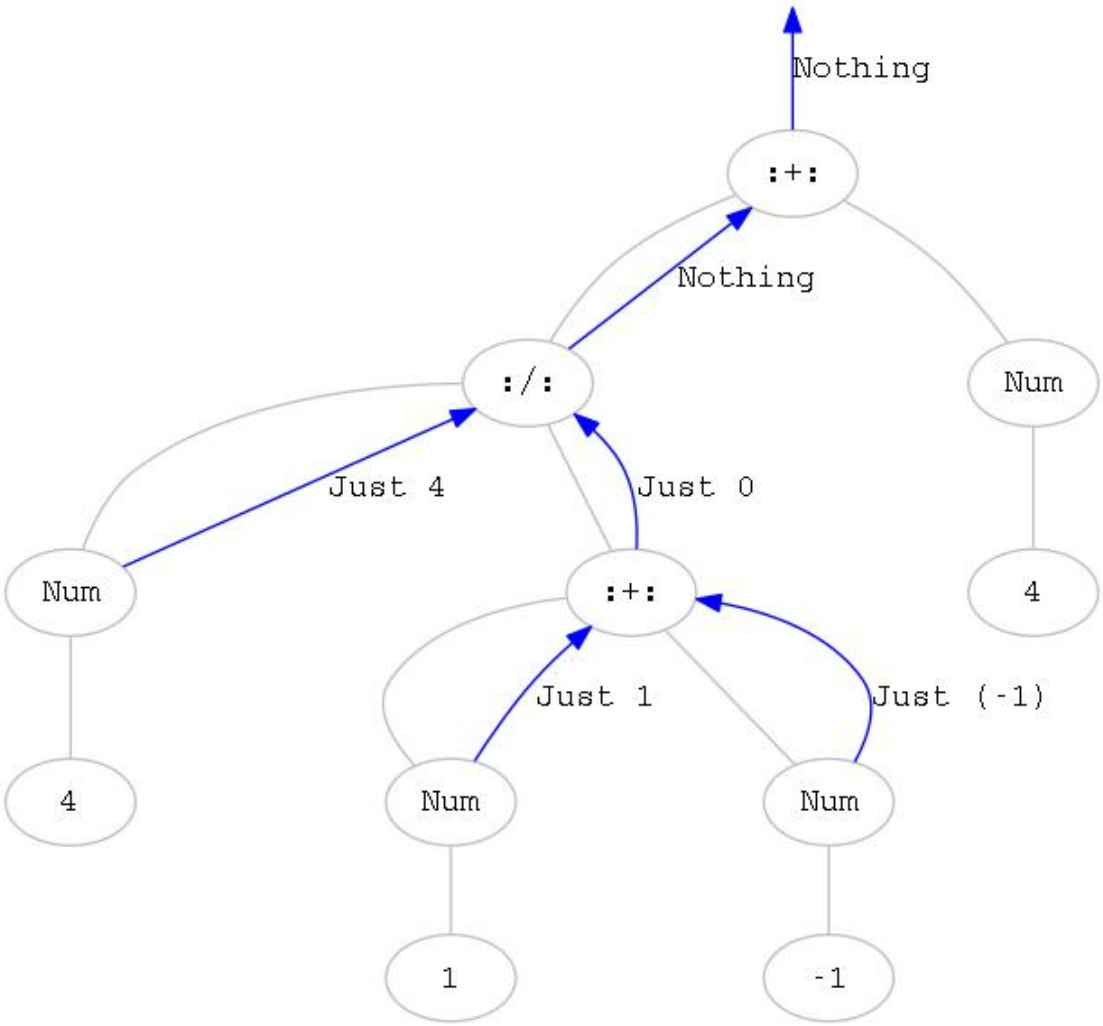

Figure 1: Bild von einem Baumdurchlauf

With this pattern, it is now possible, to stop the computation immediately, if a division by zero occurs. We can ignore the continuation and directly return `Nothing`.

```
evalCPS (e1 :/: e2) k =
  evalCPS e1 (\v1 ->
   evalCPS e2 (\v2 ->
    if v2 == 0 then Nothing else k (v1 / v2)))
```

This program does not contain any pattern matching against `Nothing`. Hence, no values are successively passed up to the root of the expression. In the error case, the program directly returns `Nothing`.

An alternative implementation of the last rule, computes the divisor first, which optimises the code such that the dividend is not evaluated, if the divisor evaluates to zero:

```
evalCPS (e1 :/: e2) k =
  evalCPS e2 (\v2 ->
   if v2 == 0 then Nothing
    else evalCPS e1 (\v1 -> k (v1 / v2)))
```

The computation of `evalCPS` with the expression from above, requires an additional continuation of type `Int -> Maybe Int`, which should by `Just`.

```
ghci> let zero = Num 1 :+: Num (-1)
ghci> let expr = (Num 4 :/: zero) :+: Num 4
ghci> evalCPS expr Just
Nothing
```

We evaluate the computation step-by-step:

```
  evalCPS expr Just
= evalCPS ((Num 4 :/: zero) :+: Num 4) Just
= evalCPS (Num 4 :/: zero) (\x ->
   evalCPS (Num 4) (\y -> Just (x+y)))
= evalCPS zero (\b ->
   if b==0 then Nothing
    else evalCPS (Num 4) (\a ->
     evalCPS (Num 4) (\y -> Just ((a/b)+y))))
= evalCPS (Num 1 :+: Num (-1)) (\b ->
   if b==0 then Nothing
    else evalCPS (Num 4) (\a ->
     evalCPS (Num 4) (\y -> Just ((a/b)+y))))
= evalCPS (Num 1) (\c ->
   evalCPS (Num (-1)) (\d ->
    if (c+d)==0 then Nothing
     else evalCPS (Num 4) (\a ->
      evalCPS (Num 4) (\y ->
       Just ((a/(c+d))+y)))))
```

```
    = evalCPS (Num (-1)) (\d ->
       if (1+d)==0 then Nothing
        else evalCPS (Num 4) (\a ->
         evalCPS (Num 4) (\y ->
          Just ((a/(1+d))+y))))
    = if (1+(-1))==0 then Nothing
       else evalCPS (Num 4) (\a ->
        evalCPS (Num 4) (\y ->
         Just ((a/(1+d))+y)))
    = Nothing
```

The result `Nothing` is directly returned and not passed through any tree structure.

The presented function `evalCPS` is more efficient than the function `eval` in the `Maybe`-Monad, but less readable. As a next step we define a monadic variant of CPS on `Maybe`-values.

Inspired by the type of `evalCPS`, we define the follwoing data type:

```
newtype CMaybe r a = CMaybe ((a -> Maybe r) -> Maybe r)
```

This type relates to the result type of `evalCPS`, where `Int` is generalized to the type parameters and `a`. Values of type `CMaybe` can be converted into common `Maybe`-values by applying them to the continuation `Just`:

```
fromCMaybe :: CMaybe a a -> Maybe a
fromCMaybe (CMaybe ca) = ca Just
```

In the converting step, the type parameters `r` and `a` get unified. Before the conversion, the type `r` usually stays polymorphic.

As a next step we define an instance of `Monad` for `CMaybe r`:

```
instance Monad (CMaybe r) where
  return x = CMaybe (\k -> k x)

  CMaybe ca >>= f = CMaybe (\k ->
    ca (\x -> let CMaybe cb = f x in cb k))
```

The implementation of `return` passes the argument to the continuation and the implementation of bind nests the computation of the second argument into the continuation of the first one. As a consequence the type of the result is of type `Maybe b`, but the argument is of type `a`. In contrast to the pure `Maybe`-Monad `(>>=)` is defined without pattern matching.

We proof the monad laws, independently of the result type of the continuation. Again for readability, we ignore then `newtype` constructor:

```
      return x >>= f
    = (\k -> k x) >>= f
```

```
  = (\k' -> (\k -> k x) (\x' -> f x' k'))
  = (\k' -> (\x' -> f x' k') x)
  = (\k' -> f x k')
  = f x

    ca >>= return
  = (\k -> ca (\x -> return x k))
  = (\k -> ca (\x -> (\k' -> k' x) k))
  = (\k -> ca (\x -> k x))
  = (\k -> ca k)
  = ca

    (ca >>= f) >>= g
  = (\k -> ca (\x -> f x k)) >>= g
  = \k' -> (\k -> ca (\x -> f x k))
           (\y -> g y k')
  = \k' -> ca (\x -> f x (\y -> g y k'))
  = \k' -> ca (\x -> (\k ->
              f x (\y -> g y k)) k')
  = \k' -> ca (\x -> (f x >>= g) k')
  = ca >>= \x -> f x >>= g
```

Both, the implementation of the monad operations and the proofs of the monad laws are independent of the result type of the continuation `Maybe r`. Later, we will define further continuation monads with different result types, for which, this implementation as well as the proofs are identical.

Now we define an instance of class `MonadPlus` for `CMaybe` , by lifting the corresponding operations from the type `Maybe`,

```
instance MonadPlus (CMaybe r) where
  mzero = CMaybe (\_ -> mzero)

  CMaybe ca `mplus` CMaybe cb =
    CMaybe (\cont -> ca cont `mplus` cb cont)
```

Again, we proof the relevant laws, ignoring the `newtype` constructors.

```
    mzero >>= f
  = (\_ -> mzero) >>= f
    \k -> (\_ -> mzero) (\x -> f x k)
  = \k -> mzero
  = mzero

    (ca `mplus` cb) >>= f
  = \k -> (ca `mplus` cb) (\x -> f x k)
```

```
    = \k -> (\k' -> ca k' `mplus` cb k') (\x -> f x k)
    = \k -> ca  (\x -> f x k) `mplus` cb (\x -> f x k)
    = \k -> (\k' -> ca (\x -> f x k')) k `mplus` (\k' -> cb (\x -> f x k')) k
    = \k -> (ca >>= f) k `mplus` (cb >>= f) k
    = ca >>= f `mplus` cb >>= f
```

Hence, the monad `CMaybe` fulfils the distributivity law, in contrast to the `Maybe`-Monad. The reason is, that the whole remaining computation is passed down with the continuation. Hence, it is not necessary to choose in the implementation of `mplus`. If one branch fails, the other one is chosen, even if we branch or fail in dependence of the monadic result.

As a consequence, we can use the monad for search and back tracking.

```
ghci> let a = return False `mplus` return True
ghci> let b = a >>= guard
ghci> b :: Maybe ()
Nothing
ghci> fromCMaybe b
Just ()
```

Now, we can define the monadic implementation of `eval`

```
eval :: MonadPlus m => Expr -> m Int
eval (Num x) = return x
eval (a :+: b) =
  do x <- eval a
     y <- eval b
     return (x+y)
eval (a :/: b) =
  do y <- eval b
     guard (y/=0)
     x <- eval a
     return (x/y)
```

Executing the program within the `CMaybe`-Monad, evaluates our expression similar to the function `evalCPS`.

The `Maybe`-Monad is not the only continuation based monad. Replacing `Maybe` by `DList` within the definition of `CMaybe`, results in a continuation based list monad, which we will inspect later.

We saw, that our implementation fulfils the following law:

```
(a `mplus` b) >>= f  =  (a>>=f) `mplus` (b>>=f)
```

although `mplus` for the monad `Maybe` does not fulfil it. The `Maybe` instance instead fullfills the law:

```
return x `mplus` a  =  return x
```

which is a law, that reminds us to a `catch` within an error-monad: only if the left argument fails, the rigth argument is executed. Such a function is often also called `orElse` and sometimes useful.

Can we define a function `orElse` for `CMaybe`, which fulfils the law:

```
return x `orElse` a  =  return x
```

This is possible, with the following code:

```
orElse :: CMaybe a a -> CMaybe a a -> CMaybe r a
CMaybe ca `orElse` CMaybe cb =
  CMaybe (\k -> (ca Just `mplus` cb Just) >>= k)
```

This implementation passes `Just` as continuation to both alternatives and combines both results by the function `mplus` from the `Maybe`-monad. The result is again converted into continuation passing style and applied to a given continuation `k`. In the type of `orElse` the type parameter of the arguments `r` gets unified with the type parameter `a` by applying `ca` and `cb` to the continuation `Just :: a -> Maybe a`.

By changing from cps, to normal style and back again, we obtain a kind of encapsulation of the computation of the first parameter of `orElse`.

The `catch` property of `orElse` can easily be shown:

```
    return x `orElse` a
  = (\k -> ((return x) Just `mplus` a Just)
           >>= k)
  = (\k -> ((\k' -> k' x) Just `mplus` a Just)
           >>= k)
  = (\k -> (Just x `mplus` a Just) >>= k)
  = (\k -> Just x >>= k)
  = (\k -> k x)
  = return x
```

Hence, `orElse` behaves similar to `mplus` in the `Maybe`-monad:

```
ghci> let a = return False `orElse` return True
ghci> fromCMaybe (a >>= guard)
Nothing
```

Using this methode, we can lift `Maybe` values to `CMaybe` values.

```
toCMaybe :: Maybe a -> CMaybe r a
toCMaybe a = CMaybe (\k -> a >>= k)
```

Here, the function `toCMaybe` is a monad homomorphismus, since the following laws hold:

```
toCMaybe (return x)  =  return x
toCMaybe (a >>= f)   =  toCMaybe a >>= toCMaybe . f
```

The proof is again simple:

```
  toCMaybe (return x)
= \k -> return x >>= k
= \k -> k x
= return x

  toCMaybe (a >>= f)  =
= \k -> (a >>= f) >>= k
= \k -> a >>= (\w -> f w >>= k) -- Ass. >>=
= \k1 -> a >>= (\x -> (f x >>= k1)) -- alpha-Konversion
= \k1 -> a >>= (\x -> (\z -> \k2 -> f z >>= k2) x k1)
= \k1 -> (\k -> a >>= k) (\x -> (\z -> \k2 -> f z >>= k2) x k1)
= (\k -> a >>= k) >>= \z -> \k2 -> f z >>= k2
= toCMaybe a >>= \z -> toCMaybe (f z)
= toCMaybe a >>= toCMaybe . f
```

Within the proofs we use the left identity and an associativity law from the underlying `Maybe`-monad.