Non-Strict Evaluation in Strict Languages

Lazy evaluation in Haskell is very useful in many cases. Especially, defining lazy data structures, like infinite lists can be useful in practice. In this chapter, we present a technique, which allows the use of non-strict evaluation within strict languages. The idea is the use of functions to suspend evaluations of data structures. For simplicity, we use functions which take an argument of type unit (`()`), since the argument is only used to suspend the evaluation.

To enforce the evaluation of a suspended a suspended function (to head normal form), we can then simply apply this function to `()`. As an example, we consider non-strict lists. The two list constructors can be represented by the following functions:

- `x : xs` is represented by `\() -> x : xs`
- `[]` is represented by `\() -> []`

In strict languages, functions are also values and a reduction underneath a lambda is not performed. Therefore, the computation is guarded from further evaluation be this construction. In this representation the second argument `xs` of the case for non-empty lists, is again supposed to be a function. Therefore it is not possible to reuse the standard list constructors. Instead we define a new data type as follows:

```
type List a = () -> ListD a
data ListD a = Nil | Cons a (List a)

-- head :: [a] -> a
headL :: List a -> a
headL xs = let (Cons y _) = xs () in y

-- tail :: [a] -> [a]
tailL :: List a -> List a
tailL xs = let (Cons _ ys) = xs () in ys

-- null :: [a] -> Bool
isNil :: List a -> Bool
isNil xs = case xs () of
             Nil -> True
             _   -> False

-- (++) :: [a] -> [a] -> [a]
app :: List a -> List a -> List a
app xs ys = if isNil xs
              then ys
              else \\() -> Cons (headL xs)
                                (app (tailL xs) ys)
```

```
-- enumFrom
from n = \() -> Cons n (from (n + 1))

-- take :: Int -> [a] -> [a]
takeL n xs =
  if isNil xs || n==0
    then \() -> Nil
    else \() -> Cons (headL xs) (takeL (n-1) (tailL xs))

showL :: Show a => List a -> String
showL xs =
  if isNil xs
    then "[]"
    else show (headL xs) ++ ":" ++ showL (tailL xs)
```

A strict evaluation of a the expression `head (from 1)` behaves as follows:

```
   head (from 1)
~> let (Cons x _) = from 1 () in x
~> let (Cons x _) = (\() -> Cons 1 (from (1 + 1)) () in x
~> let (Cons x _) = Cons 1 (from 2) in x
~> let (Cons x _) = Cons 1 (\() -> Cons 2 (from 2)) in x
~> 1
```

Note, however, that abbreviatory function definitions for `\() -> Cons x xs`, like

```
cons x xs = \x -> Cons x xs

from n = cons n (from (n+1))
```

are not possible. In a strict language, the arguments of these function will still be evaluated, before `cons` is reduced to its right-hand side and the definition for `from 1` would not terminate:

```
   isNil (from 1)
~> isNil (cons 1 (from (1+1))
~> isNil (cons 1 (from 2))
~> isNil (cons 1 (cons 2 (from (2+1)))
~> ...
```

With this technique, it is possible, to program in a non-strict manner. Unfortunately, this is less convenient than directly using a lazy language. Furthermore, this technique does not provide sharing and non-strictness is restricted to parts of your data structures. The technique has to be used in every part, which is expected to be non-strict. By default, the list elements in the definition above are still evaluated in a strict manner.

However, many strict language provide libraries for non-strict data structures. In Java

and Scala these are the Stream library. There usage is a bit more elegant, in comparison to the code from above. By means of class methods, to avoid strict evaluation, like it occurred with the definitions of a `cons` function.

For learning this technique, it is not useful to use Haskell, since mistakes will not become relevant, because of the non-strict evaluation within Haskell. You should use a strict language. Good candidate are Elm (similar syntax as Haskell), Erlang or Ruby.