An Interactive State Monad Example

Métodos de Programação I - 2005/06

Nuno F. Rodrigues
nfr@di.uminho.pt

1 Introduction

A common problem among information systems is the storage and maintenance of permanent information identified by a key. Such systems are typically known as data base engines or simply as data bases. Today the systems information market is full of solutions that provide mass storage capacities implemented in different operating system and with great amounts of extra functionalities. In this paper we will focus on the formal high level specification of data base systems in the Haskell language. We begin by introducing a high level view of a data base system with a specification of the most common operations in a functional point of view. We then augment this specification by lifting to the state monad which is then modified once again to permit input/output operations between the computations.

2 Data Base Specification

If we take a step higher in the abstraction leader when looking at any data base system, we can regard it in a data type independent manner. Thus, a data base system must supply some operational needs which are independent of whatever data structure it is used to store the actual information. In Haskell, we can specify such operational demands over the data structures by using a class like the following.

```
class BD bd where
    vazia :: bd ch inf
    acrescenta :: (Eq ch) => bd ch inf -> ch -> inf -> bd ch inf
    remove :: (Eq ch) => bd ch inf -> ch -> bd ch inf
    chaves :: bd ch inf -> [ch]
    definida :: (Eq ch) => bd ch inf -> ch -> Bool
    daInfo :: (Eq ch) => bd ch inf -> ch -> Maybe inf
```

3 Functional Specification

After having specified what our data base habitants should provide, we can provide several instances of such a system, or even expect that someone else provide them for us. A simple and inefficient instance can be given by implementing the data base as a list of pairs followed by the proper specification of the operations over this chosen data type. .

```
data LP c i = LP [(c,i)]

instance BD LP where
    vazia = LP []
    acrescenta b c i = let (LP b') = remove b c in LP ((c,i):b')
    remove (LP l) c = LP [(a,b) | (a,b) <- l, a /= c]
    chaves (LP l) = map fst l
    definida b c = c `elem` (chaves b)
    daInfo (LP l) c = case [(a,b) | (a,b) <- l, a == c] of
                        (_,i):_ -> Just i
                        _ -> Nothing
```

After having specified the above instance, we can now perform some tests over the code obtained so far. For instance, we can test the creation of a new data base followed by the storage of three records followed by a query to the information of the first introduced record.

```
dbProcedure2 :: Maybe String
dbProcedure2 = let bd1 :: LP Int String
                   bd1 = vazia
                   bd2 = acrescenta bd1 1 "A"
                   bd3 = acrescenta bd2 2 "B"
                   bd4 = acrescenta bd3 2 "C"
               in daInfo bd4 1
```

4 Introducing State

So far so good. But, there is something strange happening to the above test. Notice that we are always feeding the result of a function to another function, which can be identified as simple functional composition. Nevertheless, it isn't quite just functional composition, since functions `daInfo` or `definida` do not return any data base. Still, they do receive a data base has input.

So, what is really happening here is the passing of a data base (which will be our state) throughout the different operations that we are composing.

This pattern of the operations being applied to the data base can clearly be captured by a high order data type. Thus, we introduce the state monad, which captures this pattern of execution with the following definition.

```
data ST s l = ST { st :: s -> (s,l) }
```

Now, we have a data type that operates over a state, which in our case the data base. But, we still need a way to combine several of these operations over a state. A elegant way of providing such combinatory operators is by giving an instance of `Monad` to the `ST` data type.

```
instance Monad (ST s) where
    return x = ST (\s -> (s, x))
    f >>= g  = ST (\s -> let (s', x) = st f s
                         in st (g x) s')
```

After having defined the `ST` type and the correspondent `Monad` instance, we have to lift our previous data base operations to the new definitions in order to take advantage of the new monadic combiners. Such a lift is quite simple and can be given by the following code.

```
stVazia :: (BD b) => ST (b ch inf) ()
stVazia = ST (\_ -> (vazia,()))

stAcrescenta :: (BD b, Eq ch) => ch -> inf -> ST (b ch inf) ()
stAcrescenta c i = ST (\b -> (acrescenta b c i, ()))

stChaves :: (BD b) => ST (b ch inf) [ch]
stChaves = ST (\b -> (b, chaves b))

stRemove :: (BD b, Eq ch) => ch -> ST (b ch inf) ()
stRemove c = ST (\b -> (remove b c, ()))

stDefinida :: (BD b, Eq ch) => ch -> ST (b ch inf) Bool
stDefinida c = ST (\b -> (b, definida b c))

stDaInfo :: (BD b, Eq ch) => ch -> ST (b ch inf) (Maybe inf)
stDaInfo c = ST (\b -> (b, daInfo b c))
```

Having this state monad tool box over data bases, we can rewrite the above simple functional operations without concerns about passing the underlying state.

```
dbStProcedure :: (BD b, Num ch) => ST (b ch String) (Maybe String)
dbStProcedure = stAcrescenta 1 "A" >>
                stAcrescenta 1 "B" >>
                stAcrescenta 1 "C" >>
                stDaInfo 1

lpExec :: ST (LP ch inf) v -> v
lpExec stm = snd (st stm $ vazia)
```

Regard that in the definition of function `lpExec` which executes our state transformers over an empty data base, we have to explicitly say which instance

of the DB class we are using. In this example we use the only instance available, that is the list of pairs LP.

5 Going Interactive

Having come this far, we find ourselves able of constructing and combining operations over the data base in simple and elegant manner. But still, we can only execute such data base transformers in an atomic way, i.e., we cannot interrupt our elegantly composed operations in order to show the intermediate results nor to interact with the user.

Typically, what we want to do is to intermediate our computations over the data base with some IO operations that will let us interact with the user. In order to perform this, we have to leave our previous ST type behind and move on to a more generic type that will permit us to perform IO computations between the execution of our previous state transformers. We will call this new type STIO, with the following definition and monad instance.

```
data STIO s l = STIO { stio :: s -> IO (s,l)}

instance Monad (STIO s) where
    return x = STIO (\s -> return (s, x))
    f >>= g  = STIO (\s -> do { (s', x) <- stio f s ;
                                stio (g x) s' })
```

Again, in order to used this new type, we have to lift the previous computations that were defined over the ST monad to the new interactive STIO monad. Still, instead of having to redefine everything again, we can write a generic function that lifts every ST monad to a STIO monad and then apply it to every previous data base transformer.

```
liftST2STIO :: ST s l -> STIO s l
liftST2STIO f = STIO (\s -> return (st f s))

stioVazia :: (BD b) => STIO (b ch inf) ()
stioVazia = liftST2STIO stVazia

stioAcrescenta :: (BD b, Eq ch) => ch -> inf -> STIO (b ch inf) ()
stioAcrescenta c i = liftST2STIO (stAcrescenta c i)

stioRemove :: (BD b, Eq ch) => ch -> STIO (b ch inf) ()
stioRemove = liftST2STIO . stRemove

stioChaves :: (BD b) => STIO (b ch inf) [ch]
stioChaves = liftST2STIO stChaves

stioDefinida :: (BD b, Eq ch) => ch -> STIO (b ch inf) Bool
```

```
stioDefinida = liftST2STIO . stDefinida

stioDaInfo :: (BD b, Eq ch) => ch -> STIO (b ch inf) (Maybe inf)
stioDaInfo = liftST2STIO . stDaInfo
```

But, with these definitions we are only able to lift `ST` monads to `STIO`, still leaving us unable to introduce `IO` operations between our data base transformers.

To solve this problem we need to provide a way of lifting habitants of `IO` to `STIO`, by introducing the following lift.

```
liftIO2STIO :: IO a -> STIO s a
liftIO2STIO a = STIO (\s -> do { i <- a;
                                 return (s, i) })
```

6 Putting it All Together

Finally, we are in condition to use the above definitions to build an interactive program working over a state, which in our example will be a data base implemented as lists of pairs.

As a simple test we will define a computation (prog) that stores the value "A" into the data base; interacts with the user in order for him to introduce a new record; stores that record in the data base and terminates by printing the information stored with key 1.

```
prog :: STIO (LP String String) ()
prog = stioAcrescenta "1" "A" >>
       menuAcrescenta >>=
       uncurry stioAcrescenta >>
       stioDaInfo "1" >>=
       liftIO2STIO . print

menuAcrescenta = liftIO2STIO (do print "Introduza uma chave"
                                 c <- getLine
                                 print "Introduza informacao"
                                 i <- getLine
                                 return (c, i))
```

To execute the above test, we need to provide an interpreter that takes an interactive computation of type `STIO` and applies it to an empty data base. Such an interpreter can be given by

```
executa :: (BD b) => STIO (b ch inf) v -> IO v
executa f = do { (_, x) <- stio f vazia ;
                 return x }
```

that once applied to the above test will output the value "A".

After having understand the working mechanism of the `STIO` monad, we can define more elaborated programs with the traditional menus and user interactions by making use of the above lift functions.

7 Final Remarks

Much of what have been exposed here is already defined the libraries supplied by most of Haskell compilers/interpreters and can be used to reduce the amount of code as well as to improve the performance of the specifications.