1 Mutable Implementation

We now give the implementation of the operators seen until now with a simple mutable state.

We define our state as that of the state monad (that is a statement that evaluates to a value of type a in a state of type s has the same type of its denotational semantics):

```
data ST s a = ST(s->(a,s))
```

References will be based on the state since a reference must be easily convertible into statements, one for evaluating the reference and one for assigning it:

```
type Get s a = ST s a

type Set s a = a -> ST s ()

data Ref s a = Ref (Get s a) (Set s a)
```

We now need to represent the state (our stack). The simplest implementation of a typed stack is based on heterogeneous lists. A heterogeneous list is build based on two type constructors:

```
data Nil = Nil

data Cons h tl = Cons h tl
```

Since heterogeneous lists do not have a single type, we characterize all heterogeneous lists with an appropriate predicate:

```
class HList l

instance HList Nil

instance HList tl => HList (Cons h tl)
```

We access heterogeneous lists by index. To ensure type safety we define type-level integers, encoded as Church Numerals:

```
data Z = Z

data S n = S n


class CNum n

instance CNum Z

instance CNum n => instance CNum (S n)
```

We can now read the length of a heterogeneous list, as well as get the type of an arbitrary element of the list:

```
type family HLength l :: *

type instance HLength Nil = Z

type instance HLength (Cons h tl) = S (HLength tl)


type family HAt l n :: *

type instance HAt (Cons h tl) Z = h

type instance HAt (Cons h tl) (S n) = HAt tl n
```

We will need a way to manipulate the values of a heterogeneous list. For this reason we define a lookup predicate:

```
class (HList l, CNum n) => HLookup l n where

lookup :: l -> n -> HAt l n

update :: l -> n -> HAt l n -> l


instance (HList tl) => HLookup (Cons h tl) Z where

lookup (Cons h tl) _ = h

update (Cons h tl) _ h = (Cons h tl)


instance (HList tl, CNum n) => HLookup (Cons h tl) (S n) where

lookup (Cons _ tl) _ = lookup tl (undefined::n)

update (Cons h tl) _ v = (Cons h (update tl (undefined::n) v))
```

Now we have all that we need to instance our stack, reference and state predicates.

We begin by instancing the *Stack* predicate, since all heterogeneous lists are stacks and as such can be used:

```
instance Stack Nil where

type Push Nil a = Cons a Nil

push = Cons

pop (Cons h tl) = tl


instance (Stack tl, s ~ Cons h tl) => Stack s where

type Push s a = Cons a s

push = Cons
```

```
pop (Cons h tl) = tl
```

We instance the $Monad$ class with the ST type (as in the state monad):

```
instance Monad (ST s) where

return x = ST(\s -> (x,s))

(ST st) >>= k = ST(\s -> let (s,res) = st s in k res s)
```

We also define a way to evaluate a statement and ignoring the resulting state:

```
runST :: ST s a -> s -> a

runST (ST st) s = snd (st s)
```

Now that $Monad(STs)$ is instanced we can instance the $RefSt$ predicate for our references and state:

```
instance (HList s, n~HLength s) => RefSt Ref ST s where

eval (Ref get set) = get

(Ref get set) := v = set v

(Ref get set) *= f = do v <- get

  set (f v)

new a k =

let r_new = Ref (ST (\s -> (lookup s (undefined::n), s)))

  (\v -> ST(\s -> ((), update s (undefined::n) v)))
```

Thanks to this last instance we can now give a first working example of usage of our references with mutable state:

```
ex1 :: ST Nil Int

ex1 = 10 new (\(i :: Ref (New Nil Int) Int) ->

  do i *= (+2)

 i)


res1 :: Int

res1 = runST ex1 Nil
```

The result, as expected, is $res1 = 12$.

We complete the implementation of our system so far by adding records. We use as records heterogeneous lists to which we access via labels. A label is defined with a getter and a setter (similar to those found in the Ref constructor) as:

```
data Label r a = Label (r->a) (r->a->r)
```

We can instance the *Record* predicate:

```
instance (Stack s, HList r) => Record r Ref ST s where

type Label r a = Label r a

Ref get set <= Label read write =

Ref(do r <- get

return read r)

(\v-> do r <- get

   set (write r v))
```

To more easily manipulate records we define a function for building labels from $CNums$:

```
labelAt :: (HList l, CNum n, HLookup l n) => l -> n -> Label l (HAt l n)

labelAt _ = Label (\l -> lookup l (undefined::n)) (\l -> update l (undefined::n))
```

We can now give a second example that shows how records can be manipulated:

```
type Person = String Cons String Cons Int Cons Nil

first :: Label Person String

first = labelAt Z

last :: Label Person String

last = labelAt (S Z)

age :: Label Person Int

age = labelAt (S S Z)


mk_person f l a = (f Cons l Cons a Cons Nil)


ex2 :: ST Nil Person

ex2 = (mk_person John Smith 27) new (\(p :: Ref (New Nil Person) Person) ->

do (p <= last) *= (++  Jr.)

   (p <= age) := 25

   pv <- eval p

   return pv)
```

```
res2 :: Person

res2 = runST ex2
```

The result is, as expected, $JohnConsSmithJr.Cons25ConsNil$.

We give one last example that does not work even though at a first glance we would expect it to. This example is used to introduce the next session:

```
ex3 :: ST Nil Unit

ex3 = 10 new (\(i :: Ref (New Nil Int) Int) ->

  Hello new (\(s :: Ref (New (New Nil Int) String) String) ->

  do i *= (+2)

 s *= (++ World)

 return ())
```

This example does not even compile because:

```
i *= (+2) :: ST (New Nil Int) ()
```

while

```
s *= (++ World) :: ST (New (New Nil Int) String) ()
```

but the state monad cannot accept a state that varies between statements. It is of course worthy of notice that the above sample, though as it is does not compile, is definitely not nonsensical. Whenever we have a larger state such as

```
New (New Nil Int) String
```

we expect to be able to work with references that expect a smaller state, such as

```
New Nil Int
```

since all that is needed for them to work is contained in the larger state, and through appropriate conversion both reading and writing on the smaller state can be performed on the larger state. The notion we will use to fix this problem happens to be that of coercive subtyping.