Giuseppe Maggiore Michele Bugliesi Universit Ca' Foscari – Venice

1 Introduction

Our goal is to define a set of operators that allow us to write object-oriented programs in the Haskell language. Object orientation can be used in conjunction with various paradigms, such as (these are by no means the only possible fields of application):

- mutable programs where the state inside each object can be transparently mutated by method calls and other kinds of manipulations
- concurrent programs where the inner state of each object does not belong to the same thread, process or even machine
- reactive programs where each stateful operation is recomputed whenever the values it depends from change
- transactional programs where each stateful operation is recorded and blocks of stateful operations can be undone

We definitely wish to define our operators so that we can use anyone of these paradigms for running our object-oriented programs. For this reason we define the object oriented operators abstractly, that is inside type classes; we then proceed to give the concrete implementations that will allow us to actually run our code.

We also wish to make it simple to use our objects. One of the best ways to make some abstraction simpler to code in the Haskell language is to take advantage of monads and their syntactic sugar: we will try to make use of monads whenever possible, and we will even require so at the level of our typeclasses.

To make a working object oriented system in a type-safe and purely functional language such as Haskell we are forced to define many type-level functions and predicates. A relevant side effect of this is that all our entities are first class entities in the host language; this allows us to freely manipulate labels, method applications, and so on, and even to design a type safe reflection system.

2 Stack

We start by defining a stack typeclass, which will define the basic environment of our computations. We choose to use a stack for simplicity, and also because the state monad is much harder to make work with a heap. The stack predicate is defined as follows:

```
class Stack s where
```

```
type Push s :: * -> *
push :: s -> a -> Push s a
pop  :: Push s a -> s
```

the stack is characterized by three operators:

- a type function that takes our stack and another type as input and returns the new stack obtained by pushing the second parameter on top of the input stack
- a *push* function which works exactly as the *Push* type function but operating on values rather than types
- a *pop* function which does the opposite of what the *push* function does

3 References and Statements

We will never work directly with values, since what we are trying to accomplish requires that values are packed inside "smart" containers that are capable of doing more complex operations such as mutating a shared state, sending messages to other processes or tracking dependencies from other smart values to implement reactive updates. For this reason we will represent values in two different ways:

- as references whenever we wish to represent a pointer to some value inside the current stack
- as statements whenever we wish to represent the result of arbitrary computations

References and statements are defined respectively with one single predicate:

```
class (Stack s,Monad (st s)) => RefSt ref st s where
  eval :: ref s a -> st s a
  (:=) :: ref s a -> a -> st s ()
  (*=) :: ref s a -> (a->a) -> st s ()
  new :: s'~New s a => a -> (ref s' a -> st s' b) -> st s b
```

The *st* functor applied to the stack s is required by this definition to be a monad; thanks to this we can use our operators on references taking advantage of the syntactic sugar that Haskell offers, obtaining code that is much more intuitive to a programmer used to traditional object oriented languages.

The *eval* function takes as input a reference to a value of type a inside a stack of type s and returns a statement that will evaluate to a value of type a inside a state of type s. The $:=$ and $*=$ operators respectively assign and in-place modify

a reference and return a statement that will evaluate to a value of type unit inside a state of type s. The *new* function allocates a value of type a on the stack s, obtaining the stack s', performs some operation on this stack inside a function that takes as input the reference to the freshly allocated value of type a and returns a statement that will evaluate to a b in an s', and then returns a statement that will evaluate to the same b in an s. The *new* function is essentially a scoping function, which will:

- allocate a new value on the stack
- create a reference to this new value in the new stack
- do some work with this reference and obtain a result through the input function
- deallocate the value from the stack
- return a result that represent the result returned by the work done in the input function

An alternative definition (possibly even better) of the new operation could have been:

```
new :: s'~New s a => a -> (ref s' a -> st s' b) -> st s' b
```

which does not automatically destroy the allocated value and which instead maintains the larger stack when it returns its value. This would have worked well with a heap rather than a stack, since we could have given a heap capable of deallocating its references with a type function such as:

```
class (Heap h, Monad (st h)) => RefSt ref st h where
  Delete h -> ref h a :: *
  delete h -> ref h a -> Delete h (ref h a)
  ...
```

Sadly this approach has a major shortcoming. In fact this would make it much harder to work with monads, since at every allocation and deallocation the type of the monadic constructor would change: $(st\ h) \neq (st\ (New\ h\ a))$, and we would have been forced to define an explicit binding operator that is capable of taking care of changes in the first parameter of the statement functor:

```
explicit_bind :: st h a -> (a -> st (New h b) a') -> st (New h b) a'
```

but this would, interesting as it may be, would somewhat defeat the point of using a monad for simplicity and readability in the first place.

4 Mutable Records

We build mutable records in addition to our preceding operators. A record simply needs labels and the possiblity to (mutably) select a field from a record. Since we want to ensure mutability, we want our selection operator to take as input a reference to our record and to return as output a reference to the selected field; references can be assigned and evaluated (thanks to the $:=$, $*=$ and *eval* operators) and this is what does guarantee mutability:

```
class (RefSt ref st s) => Record r ref st s where
  type Label r :: * -> *
  (<=) :: ref s r -> Label r a -> ref s a
```

5 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 $CNum$s:

```
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.