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 Memory

We start by defining a memory typeclass, which will define the basic environment for our computations. We model our memory after a stack for simplicity. The memory predicate is defined as follows:

```
class Memory m where
  type Malloc m :: * -> *
  malloc :: m -> a -> Malloc m a
  free   :: Malloc m a -> m
  null   :: m
```

Our memory is characterized by three operators:

- a type function that takes our memory and another type as input and returns the new memory obtained by pushing the second parameter on top of the input stack
- a *malloc* function which adds a value to an existing memory
- a $free$ function which removes the last allocated value from our memory

The *null* value is a memory where all the values are null ($undefined$). This value can be used to represent a starting memory with a certain size.

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 memory
- as statements whenever we wish to represent the result of arbitrary computations

References and statements are defined respectively with the *State* predicate:

```
class (Memory m,Monad (st m m)) => State st m where
  type Ref st :: * -> *
  eval :: ref m a -> st m m a
  (:=) :: ref m a -> a -> st m m ()
  new :: a -> st m (Malloc m a) (ref (Malloc m a) a)
  delete :: st (Malloc m a) m ()
  (>>+) :: (Memory m', m'') => st m m' a -> (a -> st m' m'' b) -> st m m'' b
```

We represent state in a more complete fashion than is usually done. Usually stateful computations are represented with the type of the denotational semantics of a stateful statement where the state is the same at the beginning and end of the evaluation of the statement. In our case a stateful computation has different input and output types for the state, since memory manipulation changes the type of the state.

The *st* functor applied to the memory m twice 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. Since references are strictly linked to their kind of state, we define a type function *Ref* which returns the type of references associated to state *st*.

The *new* and *delete* operators respectively add and remove a single value from our memory.

To concatenate regular statements and state transition statements we define the $(>> +)$ operator, which is a generalized binding operator: $(>>=)$ is defined in terms of $(>> +)$ for the state monad, since $(>>=)$ simply imposes the constraint that both parameters of *st* are the same.

In our examples, we will use syntactic sugar that is not available in Haskell to make using the $(>> +)$ operator transparent; we call this the *do+* notation.

4 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 (State st s) => Record r st s where
  type Label r :: * -> *
  (<==) :: Ref st s r -> Label r a -> Ref st 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 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 s a
type Set s a = a -> St s s ()
data StRef s a = StRef (Get s s a) (Set s s a)
```

We now need to represent the state (our memory). The simplest implementation of a typed memory 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 memory, reference and state predicates.
We begin by instancing the *Memory* predicate, since all heterogeneous lists are memory and as such can be used:

```
instance Memory Nil where
  type Malloc Nil a = Cons a Nil
  malloc = Cons
  free (Cons h tl) = tl

instance (Memory tl, m ~ Cons h tl) => Memory m where
  type Malloc m a = Cons a m
  malloc = Cons
  free (Cons h tl) = tl
  null = Cons (undefined::h) (null::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 s' a -> s -> a
runSt (St st) s = snd (st s)
```

Now that *Monad*(*Sts*) is instanced we can instance the *State* predicate for our references and state:

```
instance (HList m, Memory m, n~HLength m) => State Ref St m where
  type Ref st m a = StRef m m a
  eval (StRef get set) = get
  (StRef get set) := v = set v
  (StRef get set) *= f = do v <- get
                set (f v)
  (St st) >>+ k = St(\s -> let (s',res) = st s in k res s')
  new v = let new_ref = StRef (St (\s -> (lookup s (undefined::n), s)))
```

```
                              (\v' -> St(\s -> ((), update s (undefined::n) v')))
          in St (\s -> (ref, malloc s v)
  delete = St(\s -> ((), free s)
```

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

```
ex1 :: St Nil (New Nil Int) Int
ex1 = do+ i <- new 10 :: StRef (New Nil Int) Int
          i *= (+2)
          eval 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 (Memory 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 (New Nil Person) Person
ex2 = do+ p <- new (mk_person "John" "Smith" 27) :: Ref (New Nil Person) Person
          (p <= last) *= (++ "␣Jr.")
          (p <= age) := 25
          eval p)

res2 :: Person
res2 = runSt ex2 Nil
```

The result is, as expected, "*John*" *'Cons'* "*SmithJr.*" *'Cons'* 25 *'Cons' Nil*.

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_wrong :: St Nil (Malloc (Malloc Nil Int) String) Unit
ex3_wrong = do+ i <- new 10 :: Ref (New Nil Int) Int
                s <- new "Hello" :: Ref (New (New Nil Int) String) String
                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.

6 Coercive Subtyping

We now discuss a possible solution to the problems encountered when defining the sample $ex3_wrong$. We give a predicate that expresses the relation of coercive subtyping:

```
class Coercible a b where
  coerce :: a -> b
```

We give the usual instances of this predicate, since the relation it represents is both reflexive and transitive:

```
instance Coercible a a where
  coerce a = a

instance (Coercible a b, Coercible b c) => Coercible a c where
  coerce a = coerce (coerce a)
```

7 Coercive Subtyping for References

We wish to instance the coercion predicate to references. References are:

- covariant in the referenced type
- contravariant in the state type

This happens because a reference to some a can be used whenever a reference to an a such that $a \leq a'$ is expected, and also (as seen in the third example above), a reference that works on a state s' can be used whenever a state s such that $s \leq s'$ is available. Of course, the fact that references express not only reading values and states but also writing will make this operation relatively tricky.

At the moment we will only focus on expressing the coercion relation for the state of the reference; the coercion relation for the value of the reference will be discussed together with inheritance.

The kind of operation that we wish to perform when coercing a reference to work on a larger memory is summarized in Figure 1. Whenever we wish to perform some operation on a reference to the smaller memory, we will:

- take only the first part of the (larger) input memory
- perform the operation on the obtained smaller memory through the original reference we have coerced
- replace the first part of the (larger) input memory with the (smaller) modified memory

We instance the coercion predicate for references to perform a single step of coercion, that is for the case when we have a reference to a memory tl and we want to use it where we expect a memory $Cons\ h\ tl$:

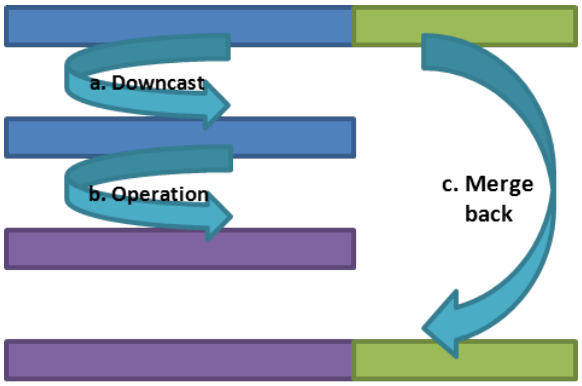

Figure 1: Coercing references.

```
class HList tl => Coercible (Ref tl a) (Ref (Cons h tl) a) where
  coerce ref =
    Ref (ST(\(Cons h tl) ->
      let (tl′,res) = get ref tl
      in (h 'Cons' tl′, res))
        (\v -> ST(\(Cons h tl) ->
          let (tl′,()) = set ref tl v
          in (h 'Cons' tl′, ()))))
  where get (Reference (ST g) _) = g
    set (Reference _ s) = \st -> \v ->
      let (ST s′) = s v
      in s′ st
```

Now we can finally rewrite the example above to make use of our new coercion operator:

```
ex3 :: ST Nil Unit
ex3 = do+ i <- new 10 :: Ref (New Nil Int) Int
          s <- new "Hello" :: Ref (New (New Nil Int) String) String
          (coerce i) *= (+2)
          s *= (++"␣World")
          return ())
```

8 Objects

We now start with the characterization of objects in our system. The *Object* predicate says that an object will be a record which supports methods and inheritance; of course all the object operators are expected to work in conjunction with the rest of the system. Since objects will reference themselves, to avoid recursive type definitions we give a predicate that allows us to freely add or remove some constructor from an object:

```
class Recursive o where
  type Rec o :: *
  to :: o -> Rec o
  from :: Rec o -> o
```

An object is required to support the *Recursive* predicate:

```
class (Record o st s, Recursive o, ro~Rec o) => Object o st s where
  type Base o :: *
  get_base :: Ref st s o -> Ref st s (Base o)

  type Method ro st :: * -> * -> *
  (<=|) :: Ref st s r -> Label r (Method ro st a b) -> a -> st s b
```

With respect to inheritance, it looks clear how we can instance coercion to take advantage (and make access more uniform) of the *base* operations:

```
instance Object o st s => Coercible (Ref st s o) (Ref st s (Base o)) where
  coerce = get_base
```

We also add a further predicate that characterizes inheritance:

```
class Inherits a b

instance (Object o st s) => Inherits o (Base o)
```

Notice that methods are defined with a different operator than the selection operator for records. This can be addressed as follows: we define a new predicate for selecting something from a reference through a label:

```
class Selectable st t s a where
  type Selection s t a :: *
  (<=) :: Ref st s t -> Label t a -> Selection s t a
```

We instance this predicate twice, one for field selection and one for method selection. Before doing so, though, we must be careful to disambiguate the last parameter in the record definition. For this purpose we add a *Field* type function that represents a placeholder type that will distinguish fields from methods and inherited types:

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

In the case of simple mutable records we can easily add a trivial constructor for *FieldSlot* such as:

```
data Field a = Field a
```

Now we can instance the selection predicate:

```
instance Record st r s => Selectable st r s (FieldSlot a) where
  type Selection s r (Field a) = Ref st s a
  (<=) = (<==)

instance (Object st o s, ro~Rec o) => Selectable st o s (Method ro a b) where
  type Selection s o (Method ro a b) = a -> st s b
  (<=) = (<=|)
```

The final operation we wish to support is that of selecting a method or a field directly from any of the inherited classes of an object without explicit coercions:

```
instance (Object st o s, bo~Base o, Selectable st bo s a) => Selectable st o s a where
  type Selection s o a = Selection s bo a
  (<=) = get_base . (<=)
```

9 Mutable Objects

We now implement mutable objects. We start with inheritance:

```
data Inherit x = Inherit x

instance (o ~ (BaseCons bo 'Cons' no), Record o ST s => Object o ST s where
  type Base o = bo
  get_base self_ref =
        StRef(do ((BaseCons base) 'Cons' tl) <- eval self_ref
           return base)
        (\base' -> do (_ 'Cons' tl) <- eval self_ref
           self_ref := ((Inherit base') 'Cons' tl)
           return ())

  ...

instance (o ~ (Unit 'Cons' so), Record o ST s => Object o ST s where
  type Base o = o
  get_base self_ref = self_ref

  ...
```

In our mutable encoding the first field of the object must be either the value of the inherited value or unit when the object does not inherit anything.

Methods enjoy the same implementation in both cases, so we just give one:

```
data MethodCons t a b = MethodCons(t -> a -> (b,t))

instance (o ~ (Unit 'Cons' so), Record o ST s => Object o ST s where
  ...

  type Method ro st a b = MethodCons ro a b
  self_ref <=| (Label read write) =
             \x -> do self <- eval self_ref
                      let (MethodCons m) = read self
                          (res,self') = m self x
                      self_ref := self'
                      return res
```

It can prove very useful to take advantage of our existing infrastructure to create methods from references and statements, so that the user of our system will not be forced to define methods by explicitly tracking mutations to the value of *this*; for this reason we add a function to the *Object* predicate that converts a method from reference to state into our format (the implementation here is the same for both instances of *Object*, so we provide only one:

```
class (Record o st s, Recursive o, ro~Rec o) => Object o st s where
  ...
  mk_method :: (StRef o o -> a -> st o b) -> Method ro st a b

instance (o ~ (Unit 'Cons' so), Record o ST s => Object o ST s where
  ...
  mk_method m = Method(\this->\args->
                         let (ST res_st) = m state_ref args
                         in res_st this)
```

10 Vectors

We now implement a simple example that shows how we can define a system of mutable vectors.

We begin by defining a 2d vector ($Vector2$) with two methods and a 3d vector ($Vector3$) with two other methods and which inherits the 2d vector:

```
type Vector2Def k = Field () 'Cons' Field Float 'Cons' Field Float 'Cons' Method k () () '
data RecVector2 = RecVector2 (Vector2Def RecVector2)
type Vector2 = Vector2Def RecVector2
instance Recursive Vector2 where
  type Rec = RecVector2
  to = RecVector2
  from (RecVector2 v) = v
x :: Label Vector2 (Field Float)
x = labelAt Z
y :: Label Vector2 (Field Float)
y = labelAt (S Z)
norm2 :: Label Vector2 (Method RecVector2 () ())
norm2 = labelAt (S S Z)
len2 :: Label Vector2 (Method RecVector2 () Float)
len2 = labelAt (S S S Z)

mk_vector2 xv yv = Field () 'Cons' Field xv 'Cons' Field yv 'Cons' mk_method (\this->\()->
                (this <= x) *= (/l)
                (this <= y) *= (/l)) 'Cons' mk_method (\this->\()->do xv <- this <= x
         yv <- this <= y
         return sqrt(xv * xv + yv * yv))

type Vector3Def k = Inherit Vector3 'Cons' Field Float 'Cons' Method k () () 'Cons' Method
```

```
data RecVector3 = RecVector3 (Vector3Def RecVector3)
type Vector3 = Vector3Def RecVector3
z :: Label Vector3 (Field Float)
z = labelAt (S Z)
norm3 :: Label Vector3 (Method RecVector3 () ())
norm3 = labelAt (S S Z)
len3 :: Label Vector3 (Method RecVector3 () Float)
len3 = labelAt (S S S Z)
instance Recursive Vector3 where
  type Rec = RecVector3
  to = RecVector3
  from (RecVector3 v) = v

mk_vector3 xv yv zv = Inherit (mk_vector2 xv yv) 'Cons' Field z 'Cons' mk_method (\this->\
                (this <= x) *= (/1)
                (this <= y) *= (/1)
                (this <= z) *= (/1)) 'Cons' mk_method (\this->\()->do xv <- this <= x
         yv <- this <= y
         zv <- this <= z
         return sqrt(xv * xv + yv * yv + zv * zv))
```

Now we can use these vectors:

```
main :: mem ~ (Vector3 'Cons' Nil) => ST Nil Nil Bool
main = do+ v <- new (mk_vector3 0.0 2.0 -1.0) :: Ref mem Vector3
           zv <- v <= x
           let v' = coerce v :: Ref mem Vector2
           zv' <- v' <= x
           (v <= norm2)()
           (v <= norm3)()
           delete
           delete
           return (zv = zv'))

res :: Bool
res = runST main Nil
```

where we expect that $res = True$. Notice how select labels that are defined for a 2d vector on an instance of a 3d vector, and how we access the same label on the value of *base* obtained through coercion on the 3d vector.

11 Alternate Implementation

In the following sections we give alternate implementations of our system. We do so in order to exploit its flexibility by showing how the same primitives allow us to define very different execution models:

- A transactional model
- A concurrent/distributed model
- A reactive model

Each model is characterized by a predicate that expresses some requirements on its state, reference or stack types so as to allow different implementations of the same model (for example to experiment with faster algorithms, etc).

The predicate for the transactional model simply states that there must be three methods for opening, closing and undoing transactions. Notice that transactions might also be used for implementing undo buffers in a very simple and clean way:

```
class Transactional st where
  beginT :: st s ()
  commitT :: st s ()
  abortT :: st s ()
```

The predicate for concurrency requires a method for forking computations:

```
class Concurrent st where
  fork :: (st s (), st s ()) -> st s ()
```

The reactive model has no specific requirements, and is simply an alternate instantiation of the object system typeclasses.

In addition to these three implementations, we will discuss a system for reflection on objects. Reflective objects will validate a predicate ($ReflectiveObject$) which allows to obtain the labels that respect certain conditions; being labels first class objects, they will then be passed around and used freely for selection on objects of the appropriate type:

```
class (Object o ref st s, ro~Rec o) => ReflectiveObject o ref st s where
  get_fields :: [Label o (Field a)]
  get_methods :: [Label o (Method ro a b)]
```