Monadic Object Encodings

Giuseppe Maggiore Michele Bugliesi
Università Ca' Foscari Venezia
Dipartimento di Informatica
{maggiore,bugliesi}@dsi.unive.it

Abstract

In this paper we define a model for expressing highly generic computations with objects in Haskell. Our objects manipulate some abstract memory. We do so with a twofold objective:

- to be able to define various, different concrete implementations of our objects that implement different behaviors such as transactional computations, reactive programming, mutable programs, etc;
- to be able to perform various kinds of static analysis on memory management and computations, thereby having type-safe dynamic memory management, type-safe reflection, and other similar opportunities.

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 State

2.1 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
  data Malloc :: * -> * -> *
  malloc :: m -> a -> Malloc a m
  free   :: Malloc a m -> m
  read   :: Malloc a m -> a
  write  :: a -> Malloc a m -> Malloc a m
```

Our memory supports type-safe allocation and deallocation thanks to two function, *malloc* and $free$, and a type function $Malloc$ that defines the type of our memory to which a value of type a is added. We also define accessors to *read* and *write* the elements of our memory.

2.2 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. Notice that we automatically associate a reference to a state *st* thanks to the type function Ref. We depart slightly from the standard representation of statements and stateful computations in that a statement in our system has different input and output types for the state, to allow the manipulation of the type of the memory to be tracked.

```
class State st where
  data Ref st :: * -> * -> *
  eval :: (Memory m) => Ref st m a -> st m m a
  (.=) :: (Memory m) => Ref st m a -> a -> st m m ()
  delete :: (Memory m) => st (Malloc a m) m ()
  new :: (Memory m) => a -> st m (Malloc a m) (Ref st (Malloc a m) a)
  (>>>=) :: (Memory m, Memory m', Memory m'') => st m m' a -> (a -> st m' m'' b) -> st m m
  (>>>) :: (Memory m, Memory m', Memory m'') => st m m' a -> st m' m'' b -> st m m'' a
```

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.

We give an evaluation operator that evaluates (dereferences) a reference into a corresponding statement and an assignment operator to assign a reference a constant value.

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.

We also define a shortcut for in-place modification of a reference:

```
(*=) :: (State st,Memory s, Monad (st s s)) => Ref st s a -> (a->a) -> st s s ()
ref *= f = do v <- eval ref
              ref .= (f v)
```

3 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):

```
class (State st) => Record r st where
  data Label st r :: * -> *
  data Field st :: * -> *
  (<==) :: Memory m => (Ref st) m r -> Label st r (Field st a) -> (Ref st) m a
a
```

4 Mutable Implementation

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

We define our state as very similar to that of the state monad (that is a statement that evaluates to a value of type a from a state of type s into 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 ()
```

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; one is for the empty list, the other will be given as a concrete constructor for the *Malloc* type function:

```
data Nil = Nil deriving (Show)

infixr `Malloc`
```

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 (Malloc 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 => 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 (Malloc h tl) = S (HLength tl)

type family HAt l n :: *
type instance HAt (Malloc h tl) Z = h
type instance HAt (Malloc 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 (Malloc h tl) Z where
  lookup (Malloc h tl) _ = h
  update (Malloc h tl) _ h' = (Malloc h' tl)

instance (HList tl, CNum n, HLookup tl n) => HLookup (Malloc h tl) (S n) where
  lookup (Malloc _ tl) _ = lookup tl (undefined::n)
  update (Malloc h tl) _ v' = (Malloc 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; we also define the single concrete constructor for the *Malloc* datatype, which thus becomes a way to create pairs:

```
instance (HList m) => Memory m where
  data Malloc a m = Malloc a m deriving (Show)
  malloc m a = Malloc a m
  free (Malloc h tl) = tl
  read   (Malloc h tl) = h
  write  h' (Malloc h tl) = Malloc h' tl
```

We instance the *Monad* class with the *St* type (as in the state monad); because of the restrictions for monads, we can only do so when the input and output states are the same:

```
instance Monad (St s s) where
  return x = St(\s -> (x,s))
  (St st) >>= k = St(\s ->
                          let (res,s') = st s
                              (St k') = k res
                          in k' 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 = fst (st s)
```

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

```
instance State St where
  data Ref St m a = StRef (Get m a) (Set m a)
  eval (StRef get set) = get
  (StRef get set) .= v = set v
  delete = St(\s -> ((), free s))
  new v = let new_ref = StRef (St (\s -> (read s, s)))
                              (\v' -> St(\s -> ((), write v' s)))
          in St (\s -> (new_ref, malloc s v))
  (St st) >>>= k = St(\s ->
                            let (res,s') = st s
                                (St k') = k res
                            in k' s')
  (St st) >>> (St st') = St(\s ->
                                 let (res,s') = st s
                                     (res',s'') = st' s'
                                 in (res,s''))
```

Notice that in the above sample a mutable reference is simply the pair of a getter and a setter function. Also, the (>> +) operator has exactly the same body as that of (>>=) that we have given for monads; this shows that, at least when embedding stateful languages inside Haskell, our definition is a generalization of the usual state monad.

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

```
ex1 :: (HList m0, m1 ~ (Malloc Int m0)) => St m0 m1 Int
ex1 = do+ i <- new 10
          i *= (+2)
          eval i)

res1 = runSt ex1 Nil

ex1' :: HList m0 => St m0 m0 Int
ex1' = do+ i <- new 10
           i *= (+2)
           eval i
           delete

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 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). We can instance the *Record* predicate:

```
instance (HList r) => Record r St where
  data Label St r a = StLabel (r->a) (r->a->r)
  data Field St a = StField a deriving (Show)
  StRef get set <== StLabel read write =
    StRef(do r <- get
             return (let (StField f) = (read r) in f))
          (\v'-> do r <- get
                    set (write r (StField v')))
```

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

```
labelAt :: forall l n . (HList l, CNum n, HLookup l n) => n -> Label St l (HAt l n)
labelAt _ = StLabel (\l -> lookup l (undefined::n))
                    (\l -> \v -> update l (undefined::n) v)
```

We can now give a second example that works with records:

```
type Person = (Field St String) `Malloc` (Field St String) `Malloc` (Field St Int) `Malloc
first :: Label St Person (Field St String)
first = labelAt Z
last :: Label St Person (Field St String)
last = labelAt (S Z)
age :: Label St Person (Field St Int)
age = labelAt (S (S Z))

mk_person :: String -> String -> Int -> Person
mk_person f l a = ((StField f) `Malloc` (StField l) `Malloc` (StField a) `Malloc` Nil)

ex2 = do+ p <- new (mk_person "John" "Smith" 27)
          (p <= last) *= (++ "␣Jr.")
          (p <= age) .= 25
          eval p

res2 = runSt ex2 Nil
```

The result is, as expected, "$John$" '$Malloc$' "$SmithJr.$" '$Malloc$' 25 '$Malloc$' 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) (New Nil Int) ()
```

while

```
s *= (++"␣World") :: St (New (New Nil Int) String) (New (New Nil Int) String) ()
```

but neither (>> +) nor (>>=) are capable of sequentializing these two statements. This said, it would not be unreasonable to expect that a statement that manipulates a smaller state such as:

```
New Nil Int
```

could be made work with references that expect a smaller state, such as

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

since all the original reference needs will be passed it together with some "trailing garbage" in the form of a larger state. The notion we will use to fix this problem happens to be that of coercive subtyping.

5 Coercive Subtyping

5.1 Coercible

We now discuss a possible solution to the problems encountered when defining the sample $ex3_{w}rong$. We give a predicate that expresses the relation of coercive subtyping:

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

5.2 Coercion 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

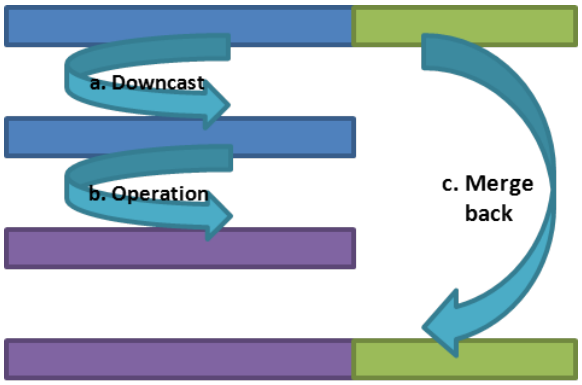

Figure 1: Coercing references.

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 $Cons\ h\ tl$:

```
instance HList tl => Coercible (Ref St tl a) (Ref St (Malloc h tl) a) where
  coerce ref =
    StRef (St(\(Malloc h tl) ->
                          let (res, tl') = get ref tl
                          in (res, h `Malloc` tl')))
          (\v -> St(\(Malloc h tl) ->
            let ((),tl') = set ref tl v
            in ((),h `Malloc` tl')))
    where get (StRef (St g) _) = g
          set (StRef _ 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 = do+ i <- new 10
          s <- new "Hello"
          ((coerce i) :: Ref St (String `Malloc` Int `Malloc` Nil) Int) *= (+2)
          s *= (++ "␣World")
          eval s

res3 = runSt ex3 Nil
```

6 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, Recursive o) => Object o st where
  data Method st :: * -> * -> * -> *
  (<=|) :: Memory s => Ref st s o -> Label st o (Method st (Rec o) a b) -> a -> st s s b
  mk_method :: (Ref st o o -> a -> st o o b) -> Method st (Rec o) a 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:

```
class Object o st => Inherits o b st where
  data Inherit :: * -> *
  get_base :: Memory s => Ref st s o -> Ref st s b
```

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 Memory m => Selectable st m t a where
  type Selection st m t a :: *
  (<=) :: Ref st m t -> Label st t a -> Selection st m 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:

```
instance (Record r st, Memory m) => Selectable st m r (Field st a) where
  type Selection st m r (Field st a) = Ref st m a
  (<=) = (<==)
```

We instance this new record typeclass as before, plus we add a simple container for fields:
Now we can instance the selection predicate:

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

7 Mutable Objects

We now implement mutable objects. We start with inheritance:

```
instance (Recursive o, Record o St) => Object o St where
  data Method St ro a b = StMethod(ro -> a -> (b,ro))
  self_ref <=| (StLabel read write) =
             \x -> do self <- eval self_ref
                      let (StMethod m) = read self
                          (res,self' :: Rec o) = m (to self) x
                      self_ref .= (from self')
                      return res
  mk_method m = StMethod(\this -> \args -> let (St res_st) = m id_ref args
                                               (res,this' :: o) = res_st ((from this) :: o)
                                           in (res,((to this') :: Rec o)))
                where id_ref = StRef (St (\s -> (s,s))) (\s' -> (St (\s -> ((),s'))))
```

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:

```
instance (Object o St, o ~ (Malloc (Inherit b) so)) => Inherits o b St where
  data Inherit a = StInherit a
  get_base self_ref =   StRef(do ((StInherit base) `Malloc` tl) <- eval self_ref
                                 return base)
                            (\base' -> do (_ `Malloc` tl) <- eval self_ref
                                          self_ref .= ((StInherit base') `Malloc` tl)
                                          return ())
```

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:

```
instance (Inherits o b st, Memory s) => Coercible (Ref st s o) (Ref st s b) where
  coerce = get_base
         in res_st this)
```

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

mk_vector2 :: Float -> Float -> Vector2
mk_vector2 xv yv = (StField xv) `Malloc` (StField yv) `Malloc` norm `Malloc` len `Malloc` l
                    where norm = mk_method (\this -> \() -> do l <- (this <= len2) ()
                                                               (this <= x) *= (/ l)
                                                               (this <= y) *= (/ l))
                                 len = mk_method (\this -> \() -> do xv <- eval (this <= x)
                                                                     yv <- eval (this <= y)
                                                                     return (sqrt(xv * xv + yv * y

type Vector3Def k = Inherit Vector2 `Malloc` Field St Float `Malloc` Method St k () () `Mal
data RecVector3 = RecVector3 (Vector3Def RecVector3)
type Vector3 = Vector3Def RecVector3
instance Recursive Vector3 where
  type Rec Vector3 = RecVector3
  to = RecVector3
  from (RecVector3 v) = v
z :: Label St Vector3 (Field St Float)
z = labelAt (S Z)
```

```
norm3 :: Label St Vector3 (Method St (Rec Vector3) () ())
norm3 = labelAt (S (S Z))
len3 :: Label St Vector3 (Method St (Rec Vector3) () Float)
len3 = labelAt (S (S (S Z)))

mk_vector3 :: Float -> Float -> Float -> Vector3
mk_vector3 xv yv zv = StInherit (mk_vector2 xv yv) `Malloc` StField zv `Malloc` norm `Mallo
                      where norm = mk_method (\this -> \() -> do l <- (this <= len3) ()
                                                                 ((get_base this) <= x) *=
                                                                 ((get_base this) <= y) *=
                                                                 (this <= z) *= (/l))
                                   len = mk_method (\this -> \() -> do xv <- eval ((get_base this)
                                                                       yv <- eval ((get_base this)
                                                                       zv <- eval (this <= z)
                                                                       return (sqrt(xv * xv + yv *

ex4 :: forall mem . mem ~ (Vector3 `Malloc` Nil) => St Nil Nil Bool
ex4 = do+ v <- new (mk_vector3 0.0 2.0 (-1.0))
          xv <- eval ((get_base v) <= x)
          let v' = coerce v :: Ref St mem Vector2
          xv' <- eval (v' <= x)
          (v' <= norm2)()
          (v <= norm3)()
          return (xv == xv')
          delete

res4 :: Bool
res4 = runSt ex4 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.

9 Alternate Implementations

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)]
```