A Monadic Object Encoding

Giuseppe Maggiore and Prof. Michele Bugliesi  
Universitá Ca’ Foscari – Venezia  
Dipartimento di Informatica

# Note su questa bozza

1. É fatta in word, ma solo perché sono piú svelto a lavorarci ☺
2. La prima conversione word 🡺 latex é automatica, e poi lavoreró (sigh) solo sulla versione latex
   1. E comunque ho scoperto che [ACM SIGPLAN accetta submissions in Word](http://www.sigplan.org/authorInformation.htm) ☺
3. Dove sono perplesso su ció che ho scritto ho *sottolineato ed evidenziato*

# Abstract

We present a novel approach for encoding mutable objects in a pure functional language. We show how our approach removes the need for problematic tail types, and thanks to an appropriate monad it makes it very simple to define, use and upcast mutable objects that behave exactly like we have come to expect in years of object oriented languages.

# Introduction

In this paper we describe a novel approach to object encoding in the pure functional language Haskell. We mainly focus on a monadic approach to object encoding in order to represent clean mutable objects. We start by discussing how we could implement mutable records, since these will be the basis of our mutable objects. Then we discuss our custom implementation of references that allow us to gain the capability of representing the computation that will lead to an object of a certain type rather than manipulating directly values of the type of our objects. At this point we show how we can build a monad that allows us to manipulate these references cleanly and implicitly. We complete our description of our monadic object encoding by defining methods and method invocation.

Since the great strides done in the last decades by object orientation researches, it is now expected by any encoding of objects that:

* Some degree of implicit and/or explicit polymorphism is supported
* There is no overhead for inheritance
* Optional multiple inheritance is supported

In light of this consideration, we discuss in relative depth how the ubiquitous shapes benchmark is implemented in our encoding and how this implementation reflects the capabilities of our system with respect to the aspects listed above. Also, we use this chance to compare our object encoding with various other object encodings that have been studied for the Haskell language.

Finally, we discuss a few extensions that we hope to add to our encoding in the future: monad transformers, extensible objects, and some other features.

# Our Object Encoding

## Mutable Records

Objects can be seen as a heterogeneous list of values (fields) and functions (methods). Of course the treatment of fields and methods is homogeneous, since in a functional language functions are first class values. This means that records are sufficient to represent objects. The default implementation of tuples and records though is too limited and does not suit our needs, especially since to represent mutable records we need setter functions to be automatically defined. For this reason we use a reduced version of the HList library as the basis for our object encoding. These records offer enough features to completely remove the need for the developer to write any kind of boilerplate code such as setters or additional type classes that make his or her records compatible with our object encoding.

The Record class defines a record inductively as the smallest set of values that are either the empty record or an application of the AddField constructor to an existing record:

data EmptyRecord = EmptyRecord deriving (Show, Eq)

data AddField e l = AddField e l deriving (Show, Eq)

infixr 8 :\*

type e :\* l = AddField e l

class Record e

instance Record EmptyRecord

instance Record l => Record (AddField e l)

Labels are Chuch Numerals: we will use these as indices for accessing records:

data Z = Z

data S a = S a

class CNum e

instance CNum Z

instance CNum a => CNum (S a)

We define the HasField n e l predicate in a manner such that:

1. n is a Church Numeral (we use its type for indexing, not its value: this way no index out of range Errors can occur)
2. l is the type of the record being looked up
3. e is the type of the n-th element of the record l

The HasField predicate ensures that we can:

1. lookup the n-th element of type e from a record l with the .! operator
2. update the n-the element of record l with a new value of e, obtaining a new record of type l with the .@ operator

class CNum n => HasField n e l where

(.!) :: l -> n -> e

(.@) :: (l,n) -> e -> l

instance HasField Z e (e :\* l) where

(AddField e \_) .! Z = e

((AddField e l),Z) .@ e' = (AddField e' l)

instance HasField n e l => HasField (S n) e (e' :\* l) where

(AddField \_ l) .! (S n) = l .! n

((AddField e l),(S n)) .@ e' = (AddField e ((l,n) .@ e'))

We are now only missing a way to build our records. First we define our labels as numerals, with the firstLabel and nextLabel functions

firstLabel = Z

nextLabel = S

Then we assign each label its value (we should not discard the label so that we can check the index of the label with the length of the record we are assembling it with) via the .= operator:

infixl 9 .=

(.=) :: CNum n => n -> e -> e

\_ .= e = e

We take assigned labels (which should have type (e,n) and not just e) and extend the empty record by adding labels to it with the .\* operator:

infixr 8 .\*

(.\*) :: e -> r -> (e :\* r)

e .\* r = AddField e r

We can build a simple record with an integer key and a string name quite easily:

{- Example -}

key = firstLabel

name = nextLabel key

test :: Int :\* String :\* EmptyRecord

test = key .= 10

.\* name .= "Pippo"

.\* EmptyRecord

We can now lookup values with the .! operator:

res :: Int

res = test .! key –-res = 10

We can also update our record with the .@ operator:

res1 :: Int :\* String :\* EmptyRecord

res1 = (test,key) .@ 20 --res1 = AddField 20 (AddField “Pippo” EmptyRecord)

## References

We need to address the fact that for writing programs that manipulate records in a mutable fashion, like:

v = (test <= key)

test <= (v+1)

then we must renounce manipulating directly values of type record. We choose to address the problem by recurring to a notion of Reference. A Reference represents a memory location inside a particular state. The state is the current environment, which can also be seen as the context (or “this”); each reference represents a value that is obtained by projecting (zero, once or many times) the environment.

A Reference denotes a way to compute a value of type inside our environment of type. This value can either be gotten or set; the accessor functions both have a signature that is quite close to the State monad, and also to the denotational semantics of imperative statements. Both these viewpoints are relevant. Reference will be used as a monadic type constructor, and it will behave in a way that is very similar to the State monad. Reference can also be understood in terms of the denotational semantics of a (nested) projection of type starting from the environment. We also define an additional type of reference, the Constant: a constant represents a computation that will return a value of type for which it does not make sense a set operation. The environment will typically be a record:

type Getter s a = s->(a,s)

type Setter s a = s->a->((),s)

data Reference s a = Reference (Getter s a) (Setter s a)

data Constant s a = Constant (Getter s a)

We define a class Value on a functor f that sums up the three most basic operations that we can perform on a reference or on a constant:

1. obtain its getter
2. turn a reference that manipulates a field of type into a reference that manipulate a field of type, given a label n such that is of type
3. build a reference from a function that represents the computation that leads to said reference
4. *promotes a constant reference into a full reference: while this risks creating references where the setter does not actually set the input value, the advantage is that we can treat all our references uniformly (non sono convinto al 100% di questa cosa)*

class Value f where

getter :: f s a -> Getter s a

(<--) :: (CNum n, HasField n b a) => (f s a) -> n -> (f s b)

build :: (s -> (f s a,s)) -> f s a

from\_constant :: (Constant s a) -> f s a

Obtaining the getter from a constant is clearly trivial:

instance Value Constant where

getter (Constant getter) = getter

Building a new constant by selection on its internal value basically requires to define a new getter that:

1. invokes the original getter, obtaining a value of type
2. returns the selection of the value of type from the value obtained from point 1)
3. passes on the modified state that comes from invoking the getter (which may perform some kind of operation on the state, because a getter represents an operation and not just a set of nested memory reads)

notice that this operation looks very similar to a binding:

(<--) :: (CNum n, HasField n b a) => (f s a) -> n -> (f s b)

all it would take to turn <-- into a full-fledged binding would be to give a (reasonable) definition of n as instead of just making it “dumbly” carry the information in its type. More on this in the future work section:

(<--) (Constant getter) n =

(Constant (\s ->

let (v,s') = getter s

in ((v .! n),s')))

building a reference from a function that takes a state and returns a reference and a (possibly) modified state is required to abstract the creation of our references inside a monad. Building simply creates a setter that invokes the argument f to obtain a constant reference and a state, and then it invokes the getter of the resulting constant reference on the resulting state:

build f = Constant(\s ->

let ((Constant getter),s') = f s

in (getter s'))

finally, creating a constant reference from a constant reference is accomplished via the identity function:

from\_constant c = c

We define the same operators on non-constant references. We start with our getter:

instance Value Reference where

getter (Reference getter \_) = getter

Then we proceed to our conversion operator (<--) so that the resulting reference will have

1. a getter which will simply invoke the selection operator .! on the result of the getter of the input reference
2. a setter which will
   1. invoke the getter of the input reference
   2. update the obtained result (which has type) with the update operator .@
   3. return the result of setting the modified result of point b) into the current state

Notice once more that both getter and setter must propagate the changes of the state after each operation. This is very consistent with the state monad, to the point where we could consider writing the getter and the setter with it. Also, the setter looks like a nested state monad: one which manipulates a state of type, the setter itself, and one which manipulates a state of type (the v which becomes v’ value). This suggests that our work could be further simplified by further “monadizing” it:

(<--) (Reference getter setter) n =

(Reference (\s ->

let (v,s') = getter s

in ((v .! n),s'))

(\s -> \x ->

let (v,s') = getter s

v' = (v,n) .@ x

((),s'') = setter s' v'

in ((),s'')))

building a reference from a function f that from a state returns a reference and a new state is done by wrapping the getter and the setter returned by invoking f on some state and invoking their respective getters and setters:

build f = (Reference (\s ->

let ((Reference getter \_),s') = f s

in (getter s'))

(\s -> \x ->

let ((Reference \_ setter),s') = f s

in (setter s' x)))

creating a reference from a constant reference is simply done by generating a setter function that does not set anything:

from\_constant (Constant getter) = (Reference getter (\s -> \x ->

let (\_,s') = getter s

in ((),s')))

we also define a simple utility to extract the setter from a reference:

setter (Reference \_ setter) = setter

The most important operator defined in the Value class is the (<--) operator: it allows us to select a field from inside the result of a reference into a resulting reference, and as such it behaves in a way that is extremely similar to the dot (.) operator found in mainstream object-oriented programming languages such as C++, Java or C#; wherever in one of these languages we would have written:

x.y

We will now write:

X <-- y

*We have avoided redefining the (.) operator because in Haskell it stands for functional composition, and also because object selection is usually represented in the literature with a right-to-left arrow such as .*

The other crucial operator that we define when working with references is the assignment operator (=:). This operator takes a Reference, a value and it returns a Reference , which makes sense because it is always possible to project the unit value from any environment; furthermore, since an operation that returns a value of type is represented with a Reference (or a Constant), then being the result of an assignment of type () we will represent it with a value of type Reference . The assignment operator will simply create a constant reference which getter invokes the setter of the input reference, with the value being set stored in the closure of the created reference:

infixl 9 =:

(=:) :: Reference s a -> a -> Reference s ()

(Reference getter setter) =: x = from\_constant(Constant(\s -> setter s x))

In our system we never need to explicitly create a reference. Indeed, this would make not much sense because it would place on the shoulders of the developer the additional burden of having to manage his references. To try and gain the advantages of having a system of mutable records through our references, but also allowing our references to be used in a fully implicit fashion, we provide the “entry point”: a ready-made basic reference called this of type Reference that represents a reference of the environment to itself:

this :: Reference s s

this = Reference (\s -> (s,s)) (\s -> \x -> ((),x))

*Examples of usage of references?*

## Reference Monad

The system of references that we have just defined is not enough for having mutable records. We need to allow a developer to seamlessly plug these references into existing Haskell programs, so that code that acts on a reference looks very much the same as if it were acting on the value represented by the reference. To obtain this goal we define a monad that acts on constructors that are instances of the Value type class.

To create our monad, we begin by defining a bind function that acts on values rather than references. The bind function takes as input a value which must be computed and which will return a value of type, and the continuation of the program which is a function that requires the previous instruction to be computed to be able to give its result. Bind invokes the build constructor (which is defined in the Value class) and creates the resulting value with a function that

1. invokes the getter of the current value, thereby computing its result
2. invokes the remainder of the program with the result obtained with point 1)

As usual some care was needed to correctly propagate the effects of intermediate evaluations:

bind :: (Value r1, Value r2) => r1 s a -> (a -> r2 s b) -> r2 s b

bind e k =

build(\s ->

let (v,s') = getter e s

res = k v

in (res,s'))

The unit function wraps a value of type inside a Constant:

unit :: a -> Constant s a

unit x = Constant (\s -> (x,s))

Finally we define the monad itself. Notice that this monad will force values to be references rather than constants; more on this in the Error! Reference source not found. section:

instance Value v => Monad (v s) where

(>>=) = bind

return x = from\_constant (unit x)

*Need to add examples of the monad in action.*

# Object Encodings

1. objects need
   1. methods
   2. protection of internal state
   3. some encodings use polymorphism, but the (<--) operator already provides a form of polymorphism (even though it is not the common subtyping polymorphism)
2. we want to integrate references and method invocation
3. we want objects to be accessible from inside the reference monad

## Methods

Methods in an object oriented language expose two main features. The first is the ability to be selected from an object which supports the method in question, in a manner similar to that of the operator (<--). The second feature that methods offer is the ability to be invoked by passing it a value of the method input type, thereby generating a value of the output type.

type Method s' a b = (a -> Reference s' b)

mk\_method :: forall a b s s' . (s'->s) -> (s->s') -> (a -> Reference s b) -> Method s' a b

mk\_method convert lift m =

\(x :: a) ->

let (Reference getter setter) = m x

in

(Reference (\(s :: s') ->

let s' = convert s :: s

(v,s'') = getter s' :: (b,s)

in (v,lift s'') :: (b,s'))

(\(s :: s') -> \(x :: b) ->

let s' = convert s :: s

(v,s'') = getter s' :: (b,s)

((),s''') = setter s'' x :: ((),s)

in ((),lift s''')))

(<<-) :: forall a s a' b c n . (CNum n, HasField n (Method a' b c) a) => (Reference s a,(a'->a),(a->a')) -> n -> (b -> Reference s c)

(<<-) (Reference get set,convert,lift) n =

\(x :: b) ->

from\_constant( Constant(\(s :: s)->

let (v,s') = get s :: (a,s)

m = v .! n

ry = m x :: Reference a' c

(y,v') = getter ry (lift v) :: (c,a')

v'' = convert v'

((),s'') = set s' v''

in (y,s'')))

## Small Client Code Sample

val = firstLabel

incr = nextLabel val

--instance Convert (Counter RecCounter) RecCounter where

convert (RecCounter r) = r

type Counter k = (Integer :\* Method k () () :\* EmptyRecord)

data RecCounter = RecCounter (Counter RecCounter)

m = mk\_method convert RecCounter

(\() ->

(do v <- ((this <-- val) :: Reference (Counter RecCounter) Integer)

(this <-- val) =: (v+1) :: Reference (Counter RecCounter) ()

return ()))

mk\_test :: Integer -> RecCounter

mk\_test i =

RecCounter( val .= i

.\* incr .= m

.\* EmptyRecord)

RecCounter test' = mk\_test 0

res2 :: Reference (Counter RecCounter) Integer

res2 = do ((this,convert,RecCounter) <<- incr) () :: Reference (Counter RecCounter) ()

((this,convert,RecCounter) <<- incr) () :: Reference (Counter RecCounter) ()

v <- (this <-- val)

return v

count = fst (getter res2 test')

Of course count has value 2.

# Shapes Benchmark

An important benchmark for object encodings is that of representing a hierarchy of shapes. The shapes problem is significant because it helps understand how the object system in question solves the most relevant issues that arise when dealing with objects:

1. how does inheritance work
2. how do we perform upcasting

We believe that our system answers quite well to these two fundamental aspects of object-oriented programming. Notice in fact that the (<--) operator is actually a coercion operator that turns a reference (an instance of an object) into another reference contained in the value returned by the first. The returned reference can be used in a mutable fashion: modifying it will modify the original, containing object as well. As such, the (<--) operator can be seen as a casting operator, where a record inherits all the types (objects or values) that it exposes as public fields. Our issues are thus solved: inheritance is multiple and it works by containing as public fields the inherited types; casting is explicit and is performed via the (<--) operator.

Let us dive in our example code.

We define a shape as a position (x and y), a move method that takes the new position, and an abstract draw method that returns a string (the result of drawing):

type Shape k = Integer :\* Integer :\* (Position -> ((),k)) :\* (() -> (String,k)) :\* EmptyRecord

data RecShape = RecShape (Shape RecShape)

being the draw method abstract, it is passed to the shape constructor and invoked directly as the body of the draw method. We could also give the internal draw function access to the private fields x and y, thus making them protected rather than private, but this is not strictly necessary:

mk\_shape :: (Position,(() -> String)) -> RecShape

mk\_shape (p@(xv,yv),draw\_f) =

RecShape( x .= xv

.\* y .= yv

.\* move .= (\p -> ((), mk\_shape(p,draw\_f)))

.\* draw .= (\() -> (draw\_f (), mk\_shape(p,draw\_f)))

.\* EmptyRecord)

we also define a function that removes the recursive constructor for invoking methods with the (<<-) operator:

shape\_flatten (RecShape r) = r

we also define a circle as a shape plus a radius. Notice that we define the Circle datatype as the radius first and the inherited shape second. If we did the opposite, that is we appended the Integer :\* EmptyRecord record to the Shape RecShape record, then we would achieve implicit casting because all the shape labels could be used on circles. The way the Circle datatype is written requires that accessing the circle position explicitly passes through the shape; of course a developer is not forced in any way into using one or the other approach.

Notice that the draw function that is passed to the shape constructor is the right drawing function for a circle:

type Circle k = Integer :\* Shape RecShape :\* EmptyRecord

data RecCircle = RecCircle (Circle RecCircle)

mk\_circle :: (Position,Integer) -> RecCircle

mk\_circle (p@(xv,yv),r) =

RecCircle( radius .= r

.\* shape\_of\_circle .= shape\_flatten (mk\_shape (p, (\() -> "(circle at " ++ show xv ++ ", " ++ show yv ++ ")")))

.\* EmptyRecord)

circle\_flatten (RecCircle r) = r

next we define a rectangle as a shape plus a width and a height, with a specific draw function:

type Rect k = Integer :\* Integer :\* Shape RecShape :\* EmptyRecord

the last object we define is the state of our application, that is a container that holds both a rectangle and a circle:

type State k = Rect RecRect :\* Circle RecCircle :\* EmptyRecord

we now define a simple function that extracts a list of shapes from a state. Notice that the casting operations arise from the use of the (<--) operator, in particolar when writing:

1. cs = c <-- shape\_of\_circle
2. rs = r <-- shape\_of\_rect

shapes :: [Reference (State RecState) (Shape RecShape)]

shapes =

let c = this <-- circle

cs = c <-- shape\_of\_circle

r = this <-- rect

rs = r <-- shape\_of\_rect

in [rs,cs]

At this point we can write a function that invokes the draw method for a list of shapes (with an accumulator, but that is hardly relevant):

draw\_many :: [Reference (State RecState) (Shape RecShape)] -> String -> Reference (State RecState) String

draw\_many [] acc = return acc

draw\_many (s : ss) acc =

do s <- ((s,shape\_flatten) <<- draw) .!! ()

draw\_many ss (s ++ acc)

We feel it is worthy to point out that no internal knowledge of how references work is needed to write this code, because the appropriate monad takes care of all the plumbing. This might be considered strong evidence of the correct working of our monad.

Finally, we can invoke the draw\_many function:

draw\_all\_shapes = fst(getter (draw\_many shapes "") state0)

As expected, the value of draw\_all\_shapes is "(circle at 50, 50)(rect at 0, 0)".

# Comparison with other Encodings

There is a vast literature that deals with object encodings in functional language, and a comparison with all that original work would fall far outside the scope of this paper. Instead we narrow our focus on:

1. the *presentation* “Object encoding in Haskell”, which lists many interesting approaches to the problem of encoding objects in a pure functional manner
2. the O’Haskell project
3. the *Haskell++* project

[…] *Comparazione vera e propria*

It is worth noting that our system allows *completely* *unrestricted* inheritance. Not only can any object inherit any type (even primitive types such as integers) by just exposing a field of that type as a public field, but also an object can inherit from the same type more than once. The fact that the selection operator is actually a casting operator is very powerful indeed.

# Conclusions and Future Work

It would be extremely interesting to extend our system in various directions. The first, somewhat obvious extension would be that of removing the need to declare a storage object for all our objects (what will be the of all our references). This could be accomplished in a manner similar to the use of the mfix operator in the OO-Haskell paper. Our need to generate a new type with a new label for our system to continue working suggests that we would need a new operator that changes the current of the current reference, returning an extended. The signature for creating a new label and a new state containing a new field of type would look approximately like:

new :: (CNum n, HasField n a s’, AddTail s a s’) => a -> Reference s () -> Reference s’ n

This way we could allocate new classes on the fly instead of all at the beginning of the application.

A less obvious but still very interesting addition would be to play with monad transformers in order to combine our monad with other monads in order to extend our objects with even more advanced features such as exceptions, concurrency and many more features.

We have drafted an implementation of this system in the F# language, to start studying optimizations to our monad operators in order to remove some of the performance overhead that is encountered with all the closures and all the copies of the state that are generated. Initial benchmarks look quite promising, so we will put some more work in this area.

Extensible objects and overriding methods are very important aspects of object encodings and might not be too difficult to support given the inherent flexibility of type lists such as our records. Also, limiting extensibility and overriding is now a feature that most object-oriented languages support in order to guarantee that certain classes will work as expected, for example for security reasons.

One final point is that we would like to further explore the issue of the need for explicit type signatures. It is in fact important to notice that the samples listed in our code will not compile in the *latest version* of the GHC compiler, even though they are correctly typed. This happens because the type inference algorithm encounters some difficulties when dealing with complex combinators libraries such as ours. This said, just by adding explicit typing information the code snippets of the paper all compile and work flawlessly.

# References

1. HList
2. Object Encodings
3. Object Encoding in Haskell
   1. Sub-references
4. Extensible objects
5. O’Haskell
6. Taming Effects
7. Monads
   1. Moggi
   2. Wadler
   3. Vari altri