# Abstract

[...]

# Introduction

[...]

# Problem Statement

We wish to adapt an object encoding to represent the following objects (expressed in a Java-like pseudo-syntax) to a pure functional language.

class A

{

    int x;

    A(int v) {x=v;}

    void incr() {x++;}

    void decr() {x--;}

}

class B inherits A

{

    int y;

    A(int v) : base(v) {y=v2;}

    void incr() {y++; base.incr();}

    void decr() {y--; base.incr();}

    int M() {base.incr(); return y;}

}

**Naive Encoding**

[...]

**Monadic Encoding**

An essential aspect of these objects is their stateful nature. It is quite clear that a naive encoding would not be able to enforce the requirements of mutability. [...]

To solve this problem, we start by observing more closely the signature that we expect our methods will have in a pure functional language:

A.incr :: Unit  A  A

A.decr :: Unit  A  A

B.incr :: Unit  B  B

B.decr :: Unit  B  B

B.M    :: Unit  B  intB

These signatures look strikingly familiar to one who has already studied monads [...].

In particular the monad these signatures remind the most is the State monad [...].

Our proposed encoding is the following:

module A

    data  = {x:int}

    x = mk\_field .x

    A v = {x=v}

    incr() =

        do v  x.get

           x.set (v+1)

    decr() =

        do v  x.get

           x.set (v-1)

module B

    data  = {y:int; base:}

    y = mk\_field .y

    base = mk\_field .base

    B v = {y=v\*2; base=A.A v}

    incr() =

        do v  y.get

           y.set (v+1)

           base A.incr()

    decr() =

        do v  y.get

           y.set (v-1)

           base A.decr()

    M() =

        do base A.incr()

           v  y.get

           return v

Let us give a look at the way we have described the structure in a pure functional setting:

1. a state monad with internal state of type  and return value  has type State
2. each class X requires an internal type  which contains the current state of the class
3. each field of type  of the internal type  has a getter and a setter with monadic signatures:
   * get :: State
   * set ::  State Unit
4. each method with input  and output  has signature  State
5. each method is a monad, and it can access the fields with their getters and setters without ever explicitly accessing the monadic state
6. each method implicitly takes "self" as a parameter of type , even though the developer is not required to write it
7. invoking a method on a field (like "base") is done through the operator , which is the equivalent of the "dot" operator usually found in object-oriented languages

The two (very important) constructs that we have left unspecified are the field constructor mk\_field and the object method invocation operator .

The field constructor works like this (WARNING: NOT ACTUAL SYNTAX):

data Field  = {get :: State ; set ::  State Unit }

mk\_field .f =

    {get = tt.f,t;

     set = f't (),{t with f = f'}}

The field constructor simply wraps in a state-monad-friendly construct a getter and a setter for a record field.

The operator applies a monadic method to a field:

self m =

    do s self.get

        v,s' = m s

        self.set s'

        return v

This operator takes as input a field, which isolates a single field with respect to the context in which it is being used. The field is read, the method is applied to it, and then the new value of the field is set back to it before returning the result of the method. The operator allows a method to both return a result (v) and modify the internal state which the field has accessors to. The signature of the operator is:

() :: Field  State  State

This suggests that the operator is actually a conversion that takes a method which acts on an object which is contained in the current (self) object, and returns a new method which performs the same operation only "lifted" [???scelta del termine???] to the level of self.

**Conversion (casting)**

A very common operation which we find in object-oriented systems is that of casting an object to another instance of itself which is less specified than the original. In our case casting will consist in taking a field which acts on an instance of a certain type and returning a field which acts on a field of the original instance.

Casting an object of type B to type A is done by adding the following method to module B:

module B

    ...

    convertToA self = {get = self b.get; set b' = self b.set b'}

The conversion function has type

convertToA :: Field  Field

Which generates an instance which (informally speaking) acts as self.base. Invoking a method on (convertToA self), like by writing (convertToA self A.incr()) will modify the instance represented by self.

**Usage in Client Code**

An object of the sort we have just defined can be used inside a state monad similar to one of the methods of the object itself:

main() =

    A.A 10 >>> [???Lo ho usato correttamente? Viene da Control.Arrow...???]

    do incr()

       incr()

       decr()

       return y.get

Would correspond (in our Java-like pseudo-syntax from above):

void main() {

    A a = new A(10);

    a.incr();

    a.incr();

    a.decr();

    return a.y;}

Which is remarkably similar to the monadic version. Note that in this case the state monad is working in a way that is very similar to the With...End construct found in the Visual Basic language [...].

Should we need to use more than one object in the same context, then we would need to define a containing object and each instance we need to use will be accessed through a field:

type  = {a:;b:}

a = mk\_field .a

b = mk\_field .b

main() =

    {a = A.A 10; b = B.B 20} >>>

    do a incr()

       b incr()

       a' = B.convertToA b

       a' incr()

       -- a' M() would not compile, as expected!

       return ()

This last example would translate, in our Java-like language, to:

main() {

    A a = new A(10);

    B b = B(20);

    a.incr();

    b.incr();

    A a' = (A)b;

    a'.incr();

}

As before, the two listings can be matched one to the other line-by-line.

**Object Extensions**

It is noteworthy to realize that this scheme allows extensible objects. We can define new methods simply by creating a function with the same signature as that of a normal method, and this new function could be used inside a monad exactly like the "regular" methods. As an example, let us consider an extension method to A which calls "incr" twice:

incr\_twice() =

    do incr()

       incr()

In a very similar manner we could add to an object any field (which would probably compute some expression) [...].

**Advantages**

[...]

**Future Work**

[...]