*Giuseppe Maggiore Michele Bugliesi Universit Ca' Foscari – Venice*

## 1 Motivation

We wish to introduce a series of constructs that allow us to write object-oriented programs that are capable of running with very different runtime behaviors with minimal to zero modification of the program. Our constructs (characterized by a type class $Heap$) are:

- a storage (heap) $h$
- a reference to a location in the storage $ref$
- an allocation operator $>> +$ for values inside the heap
- a program statement *st*
- an evaluation operator *eval*
- an assignment operator $:=$
- a selection operator for fields and methods $\Leftarrow$

We also give a subtyping and an inheritance relation for polymorphism.

Thanks to these constructs we can write generic object-oriented programs. We show how different concrete implementations of the $Heap$ class can be given that allow highly varying execution schemes.

The first execution scheme is a simple mutable program that manipulates references to $Point3D$, which inherits from $Point2D$:

```
length2d : Label Point2D (Method Point2D Unit Float)
length3d : Label Point3D (Method Point3D Unit Float)
normalize : Label Point3D (Method Point3D Unit Unit)


f : Heap h ref st, Point3D ≤ Point2D ⇒ ref h Point 3D → st h float
f p =
 do  l ← (p ⇐ length3d) ()
     (p ⇐ normalize) ()
     l' ← (p ⇐ length2d) ()
```

```
        return l / l'


v = runMutable (do (newPoint3D 1.0 2.0 −1.0) >>+ f)
```

In this example $v$ will be equal to the original length of the vector divided by the length of the $x, y$ vector after normalizing it in three dimensions.

As another example let us consider some code that performs operations on a bank account. This code makes us of an extension to our system, that is the $Transactional$ predicate on our statements that allows us to use the *beginT*, *abortT* and *commitT* statements that respectively mark the beginning of a transaction and its end by failure or success:

```
get_balance : Label Account (Method Account Unit Int)
withdraw : Label Account (Method Account Int Unit)
deposit : Label Account (Method Account Int Unit)


banker : Heap h ref st, Transactional st ⇒ [Int] → [Int] → ref h Account
banker ws ds account =
  do  begintT
        transactions ws ds account
        b ← (account ⇐ get_balance) ()
        return b
  where transactions [] [] =
          do  b ← (account ⇐ get_balance) ()
              if b < 0 then abortT
              else commitT
        transactions (w:ws) [] =
          do  (account ⇐ withdraw) w
              transactions ws []
        transactions ws (d:ds) =
          do  (account ⇐ deposit) d
              transactions ws ds
```

```
v = runMutable (do (newAccount 100) >>+ banker [10;5;700] [200;500;240])
```

where we expect that the transaction will fail and $v = 100$.

We can also express concurrent computations where assignment corresponds to sending a value and evaluation corresponds to receiving one from a network channel; the *Concurrent* predicate also allows us to use the *fork* and *sleep* statements:

```
main : Heap h ref st , Concurrent st ⇒ st h ()
main =
  0 >>+ (λchannel.
      fork (p1 channel 0)
           (p2 channel))
  where p1 channel i =
          do  channel := i -- send
              sleep 1000
              i' ← eval channel -- receive
              p1 channel (i'+1)
        p2 channel =
          do  i ← eval channel -- receive
              channel := (i*2)
              p2 channel


runConcurrent main
```

where we expect the code to never stop running while the two processes continuosly exchange messages.

Finally we show how we can express reactive programs that are capable of automatically updating certain values where those they depend from get modified; the *Reactive* predicate enables the reactive operators $! =$ for assignment and $!+$ for sum:

```
f : Num a, Heap h ref st , Reactive ref ⇒ ref h a → ref h a → ref h a → st h a
f a b c =
  do  b := 10
      c := −5
      a != b !+ c
```

```
    x ← a
    b := 5
    x' ← a
    return (x,x')

v = runReactive (do 0 >>+ (λa.0 >>+ (λb.0 >>+ (λc.f a b c))))
```

where we expect our program to return (5,0) since reassigning $b$ automatically modifies the value of $a$ which depends from it.

These are by no means the only possible applications of our system. Not only these can be combined (for example to form transactional and concurrent programs) through monad transforms, but also many more applications could be found.