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
  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.

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:

```
class (Memory m,Monad (st m)) => RefSt ref st m where
  eval :: ref m a -> st m a
  (:=) :: ref m a -> a -> st m ()
  new :: a -> st (Malloc m a) (ref (Malloc m a) a)
  (>>+) :: st m a -> (a -> st (Malloc m b) c) -> st (Malloc m b) c
```

The *st* functor applied to the memory m 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 make a new slot in our memory available by invoking the *new* function; this will return a reference to that value, after it has been properly initialized to the first parameter of *new*. We sequentialize two statements, one that works on a memory and another that works on a larger memory with the $>> +$ operator.

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:

```
do+ x <- m1
    m2
```

becomes

```
do x <- m1
   m2
```

if $m1$ and $m2$ have the same monadic type, that is $m1 :: M\ a$ and $m2 :: M\ b$; otherwise, if $m1 :: M\ a$ and $m2 :: \hat{M}\ b$, it will become:

```
m1 >>+ (\x -> do+ m2)
```

The behavior of *do+* is thus very similar to that of *do*, but it is also capable of sequentializing monadic values of different types (provided that an appropriate convert-and-bind operator such as $>> +$ is defined on the two monads).

We also add a useful operator, which can be used as a shortcut for the common eval, modify and re-assign sequence of operations:

```
(*=) :: RefSt ref st m => ref m a -> (a->a) -> st m ()
r *= f = do v <- eval r
            r := (f v)
```