# Capabilities and Coeffects

Ben Lippmeier
University of New South Wales
SAPLING 2013/12/16

# Destructive Initialization

```
timesTwo :: Array Int -> Array Int
```

# Destructive Initialization

```
timesTwo :: Array Int -> Array Int
timesTwo arr
 = unsafePerformIO
 $ do marr :: MArray Int <- M.new (I.length arr)
      loop (I.length arr)
           (\ix -> do val <- I.read arr ix
                      M.write marr ix (val * 2))
      M.unsafeFreezeMArr marr
```

# Destructive Initialization

```
timesTwo :: Array Int -> Array Int
timesTwo arr
 = unsafePerformIO
 $ do marr :: MArray Int <- M.new (I.length arr)
      loop (I.length arr)
           (\ix -> do val <- I.read arr ix
                      M.write marr ix (val * 2))
      M.unsafeFreezeMArr marr
```

# Destructive Initialization

```
timesTwo :: Array Int -> Array Int
timesTwo arr
 = unsafePerformIO
 $ do marr :: MArray Int <- M.new (I.length arr)
       loop (I.length arr)
            (\ix -> do val <- I.read arr ix
                       M.write marr ix (val * 2))
       M.unsafeFreezeMArr marr


unsafePerformIO  :: IO a -> a
unsafeFreezeMArr :: MArray a -> IO (Array a)
```

# ST monads are only a partial solution

```
runSTArray :: Ix i
           => (forall s. ST s (STArray s i e))
           -> Array i e
```

- Fine for single arrays.
- What about tuples of arrays, or general mutable data?
- ...

```
readRef  :: forall (r : Region) (a : Data)
         .  Ref r a -> S (Read r) a

writeRef :: forall (r : Region) (a : Data)
         .  Ref r a -> a -> S (Write r) Unit
```

```
readRef  :: forall (r : Region) (a : Data)
         .  Ref r a -> S (Read r) a

writeRef :: forall (r : Region) (a : Data)
         .  Ref r a -> a -> S (Write r) Unit


f :: forall (r1 : Region)
  .  Ref r1 Nat -> S (Read r1 + Write r1) Unit
```

```
readRef  :: forall (r : Region) (a : Data)
         .  Ref r a -> S (Read r) a

writeRef :: forall (r : Region) (a : Data)
         .  Ref r a -> a -> S (Write r) Unit


f :: forall (r1 : Region)
   .  Ref r1 Nat -> S (Read r1 + Write r1) Unit

f = /\(r1 : Region). \(ref : Ref r1 Nat).
    box
     do
     { x : Nat = run readRef [r1] [Nat] ref
     ; run writeRef [r1] [Nat] ref (add x x) }
```

```
readRef  :: forall (r : Region) (a : Data)
         .  Ref r a -> S (Read r) a

writeRef :: forall (r : Region) (a : Data)
         .  Ref r a -> a -> S (Write r) Unit


f :: forall (r1 : Region)
   .  Ref r1 Nat -> S (Read r1 + Write r1) Unit

f = /\(r1 : Region). \(ref : Ref r1 Nat).
    box
     do                          S (Read r1) Nat
     { x : Nat = run readRef [r1] [Nat] ref
     ; run writeRef [r1] [Nat] ref (add x x) }
```

```
readRef  :: forall (r : Region) (a : Data)
         .  Ref r a -> S (Read r) a

writeRef :: forall (r : Region) (a : Data)
         .  Ref r a -> a -> S (Write r) Unit


f :: forall (r1 : Region)
   .  Ref r1 Nat -> S (Read r1 + Write r1) Unit

f = /\(r1 : Region). \(ref : Ref r1 Nat).
    box
     do                              S (Read r1) Nat
     { x : Nat = run readRef [r1] [Nat] ref
     ; run writeRef [r1] [Nat] ref (add x x) }
                          S (Write r1) Unit
```

```
readRef  :: forall (r : Region) (a : Data)
         .  Ref r a -> S (Read r) a

writeRef :: forall (r : Region) (a : Data)
         .  Ref r a -> a -> S (Write r) Unit


f :: forall (r1 : Region)
   .  Ref r1 Nat -> S (Read r1 + Write r1) Unit

f = /\(r1 : Region). \(ref : Ref r1 Nat).
      box
       do
       { x : Nat = run readRef [r1] [Nat] ref
       ; run writeRef [r1] [Nat] ref (add x x) }
              S (Read r1 + Write r1) Unit
```

```
readRef  :: forall (r : Region) (a : Data)
         .  Ref r a -> S (Read r) a

allocRef :: forall (r : Region) (a : Data)
         .  a -> S (Alloc r) (Ref r a)
```

```
z :: Nat
z = private r with {Alloc r; Read r} in
do { x = run allocRef [r] [Nat] 4
   ; run readRef [r] [Nat] x }
```

```
readRef  :: forall (r : Region) (a : Data)
         .  Ref r a -> S (Read r) a

allocRef :: forall (r : Region) (a : Data)
         .  a -> S (Alloc r) (Ref r a)


  z :: Nat
  z = private r with {Alloc r; Read r} in
      do { x = run allocRef [r] [Nat] 4
         ; run readRef [r] [Nat] x }


f = /\(r1 : Region). \(ref : Ref r1 Nat).
    box
     do
     { x : Nat = run readRef [r1] [Nat] ref
     ; run writeRef [r1] [Nat] ref (add x x) }
```

```
readRef  :: forall (r : Region) (a : Data)
         .  Ref r a -> S (Read r) a

allocRef :: forall (r : Region) (a : Data)
         .  a -> S (Alloc r) (Ref r a)
```

**Concrete binding**

```
z :: Nat
z = private r with {Alloc r; Read r} in
    do { x = run allocRef [r] [Nat] 4
       ; run readRef [r] [Nat] x }
```

**Abstract binding**

```
f = /\(r1 : Region). \(ref : Ref r1 Nat).
    box
     do
     { x : Nat = run readRef [r1] [Nat] ref
     ; run writeRef [r1] [Nat] ref (add x x) }
```

```
h :: forall (r1 : Region)
   .  S (Alloc r1) (Ref r1 Nat)

h = /\(r1 : Region). box
    extend r1 using r2 with {Alloc r2; Write r2} in
    do { x = run allocRef [r2] [Nat] 0
    ; _ = run writeRef [r2] [Nat] x 5
    ; x }
```

```
h :: forall (r1 : Region)
   .  S (Alloc r1) (Ref r1 Nat)

h = /\(r1 : Region). box
    extend r1 using r2 with {Alloc r2; Write r2} in
    do { x = run allocRef [r2] [Nat] 0
    ; _ = run writeRef [r2] [Nat] x 5
    ; x }
```

Ref **r2** Nat

```
h :: forall (r1 : Region)
   .  S (Alloc r1) (Ref r1 Nat)

h = /\(r1 : Region). box
    extend r1 using r2 with {Alloc r2; Write r2} in
    do { x = run allocRef [r2] [Nat] 0
    ; _ = run writeRef [r2] [Nat] x 5
    ; x }
```

Ref **r1** Nat

```
g :: forall (r1 : Region)
   .  S (Alloc r1)
        (Tuple2 (Ref r1 Nat)
                (S (Write r1) Unit))
```

```
g :: forall (r1 : Region)
   .  S (Alloc r1)
        (Tuple2 (Ref r1 Nat)
                (S (Write r1) Unit))

g = /\(r1 : Region). box
     extend r1 using r2 with {Alloc r2; Write r2} in
     do { x = run allocRef [r2] [Nat] 0
        ; f = writeRef [r2] [Nat] x 5
        ; T2  [Ref r2 Nat]
              [S (Write r2) Unit]
              x f
     }
```

```
g :: forall (r1 : Region)
   .  S (Alloc r1)
        (Tuple2 (Ref r1 Nat)
                (S (Write r1) Unit))

g = /\(r1 : Region). box
     extend r1 using r2 with {Alloc r2; Write r2} in
     do { x = run allocRef [r2] [Nat] 0
        ; f = writeRef [r2] [Nat] x 5
     ; T2  [Ref r2 Nat]
           [S (Write r2) Unit]
           x f
     }

private r with {Alloc r; Read r} in
case run g [r] of
       T2 r f -> run f
```

```
g :: forall (r1 : Region)
   .  S (Alloc r1)
        (Tuple2 (Ref r1 Nat)
                (S (Write r1) Unit))

g = /\(r1 : Region). box
     extend r1 using r2 with {Alloc r2; Write r2} in
     do { x = run allocRef [r2] [Nat] 0
        ; f = writeRef [r2] [Nat] x 5
        ; T2  [Ref r2 Nat]
              [S (Write r2) Unit]
              x f
     }

private r with {Alloc r; Read r} in
case run g [r] of
       T2 r f -> run f
```

```
g :: forall (r1 : Region)
   .  S (Alloc r1)
        (Tuple2 (Ref r1 Nat)
                (S (Write r1) Unit))

g = /\(r1 : Region). box
     extend r1 using r2 with {Alloc r2; Write r2} in
     do { x = run allocRef [r2] [Nat] 0
        ; f = writeRef [r2] [Nat] x 5
        ; T2  [Ref r2 Nat]
              [S (Write r2) Unit]
              x f
     }

private r with {Alloc r; Read r} in
case run g [r] of                    S (Alloc r + Write r) Nat
       T2 r f -> run f
```

```
g :: forall (r1 : Region)
   .  S (Alloc r1)
        (Tuple2 (Ref r1 Nat)
                (S (Write r1) Unit))

g = /\(r1 : Region). box
     extend r1 using r2 with {Alloc r2; Write r2} in
     do { x = run allocRef [r2] [Nat] 0
        ; f = writeRef [r2] [Nat] x 5
        ; T2   [Ref r2 Nat]
               [S (Write r2) Unit]
               x f
     }
```

**Error: No (Write r) capability.**

```
private r with {Alloc r; Read r} in
case run g [r] of                    S (Alloc r + Write r) Nat
       T2 r f -> run f
```

# Related work

- **Andrzej Filinski**
  *Representing Layered Monads*
  “Reification and Reflection”

- **Frank Pfenning and Rowan Davies**
  *A Judgmental Reconstruction of Modal Logic*
  “Lax modality”

- **Paul Levy**
  *Call by Push Value*
  Has reification and reflection -like operators

- **Aleksandar Nanevski**
  *A Modal Calculus for Exception Handling*