

Automatically Escaping Monads

Ben Lippmeier

HIW 2016/09/23

data Color where

Red : Color

Blue : Color

```
one : Bool -> Color
```

```
one x
```

```
  = case x of
```

```
    True  -> Red
```

```
    False -> Blue
```

```
two : Ref Bool -> Color
two x
  = case readRef x of
    True   -> Red
    False  -> Blue
```

```
readRef : Ref a -> a
```

```
two : Ref Bool -> IO Color
```

```
two x
```

```
  = case readRef x of
```

```
    True  -> Red
```

```
    False -> Blue
```



```
readRef : Ref a -> IO a
```

```
three : Ref Bool -> IO Color
three x
  = do x' <- readRef x
      return $ case x' of
          True  -> Red
          False -> Blue
```

```
readRef : Ref a -> IO a
```

```
four : (Nat, Color) -> Color
four  (n, c)
  | n > 5
  , Red <- c
  = Blue

  | otherwise
  = Red
```

```
five : (Nat, Ref Color) -> IO Color
five  (n, c)
  | n > 5
  , Red <- readRef c
= Blue

  | otherwise
= Red
```



```
readRef : Ref a -> IO a
```


Run and Box

```
two : Ref Bool -> Color
two x
  = case readRef x of
    True   -> Red
    False  -> Blue
```

```
six : Ref Bool -> S Read Color
six x
  = box (case (run (readRef x)) of
          True  -> Red
          False -> Blue)
```

```
six : Ref Bool -> S Read Color
six x
  = box (case (run (readRef x)) of
          True  -> Red
          False -> Blue)
```

```
readRef : Ref a -> S Read a
```

```
six : Ref Bool -> S Read Color
```

```
six x
```

```
= box (case (run (readRef x)) of
  True  -> Red
  False -> Blue)
```



```
S Read Bool ; Pure
```

```
readRef : Ref a -> S Read a
```

```
six : Ref Bool -> S Read Color
```

```
six x
```

```
= box (case (run (readRef x)) of  
      True  -> Red  
      False -> Blue)
```



```
Bool ; Read
```

```
readRef : Ref a -> S Read a
```

```
six : Ref Bool -> S Read Color
```

```
six x
```

```
= box (case (run (readRef x)) of  
      True  -> Red  
      False -> Blue)
```




```
Color ; Read
```

```
readRef : Ref a -> S Read a
```

```
six : Ref Bool -> S Read Color
```

```
six x
```

```
= box (case (run (readRef x)) of  
      True  -> Red  
      False -> Blue)
```



```
S Read Color ; Pure
```

```
readRef : Ref a -> S Read a
```



```
run ~ safePerformIO
```

```
six : Ref Bool -> S Read Color
```

```
six x
```

```
= box (case (run (readRef x)) of
        True  -> Red
        False -> Blue)
```



```
S Read Color ; Pure
```

```
readRef : Ref a -> S Read a
```

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\Gamma, x : \tau_1 \vdash M : \tau_2$$

$$\Gamma \vdash (\lambda x : \tau_1. M) : \tau_1 \rightarrow \tau_2$$

$$\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2$$

$$\Gamma \vdash M_2 : \tau_1$$

$$\Gamma \vdash M_1 M_2 : \tau_2$$

$$x : \tau \in \Gamma$$

$$\Gamma \vdash x : \tau ; \text{Pure}$$
$$\Gamma, x : \tau_1 \vdash M : \tau_2 ; \text{Pure}$$

$$\Gamma \vdash (\lambda x : \tau_1. M) : \tau_1 \rightarrow \tau_2 ; \text{Pure}$$
$$\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 ; \sigma_1 \quad \Gamma \vdash M_2 : \tau_1 ; \sigma_2$$

$$\Gamma \vdash M_1 M_2 : \tau_2 ; \sigma_1 + \sigma_2$$

$$x : \tau \in \Gamma$$

$$\Gamma \vdash x : \tau ; \text{Pure}$$
$$\Gamma, x : \tau_1 \vdash M : \tau_2 ; \text{Pure}$$

$$\Gamma \vdash (\lambda x : \tau_1. M) : \tau_1 \rightarrow \tau_2 ; \text{Pure}$$
$$\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 ; \sigma_1 \quad \Gamma \vdash M_2 : \tau_1 ; \sigma_2$$

$$\Gamma \vdash M_1 M_2 : \tau_2 ; \sigma_1 + \sigma_2$$
$$\Gamma \vdash M : S \sigma_1 \tau ; \sigma_2 \quad \Gamma \text{ supports } \sigma_1$$

$$\Gamma \vdash \text{run } M : \tau ; \sigma_1 + \sigma_2$$
$$\Gamma \vdash M : \tau ; \sigma_1$$

$$\Gamma \vdash \text{box } M : S \sigma_1 \tau ; \text{Pure}$$

$$f : \tau_1 \xrightarrow{e} \tau_2$$

Latent effects = Bad

$$f : \tau_1 \rightarrow S \ e \ \tau_2$$

Effect indexed computation types = Good

$$f : \tau_1 \xrightarrow{e} \tau_2$$

Latent effects = Bad

$$f : \tau_1 \rightarrow S\ e\ \tau_2$$

Effect indexed computation types = Good
(and are not monads)

Automatic Insert of `run` and `box`

```
six : Ref Bool -> S Read Color
six x
  = box (case (run (readRef x)) of
          True  -> Red
          False -> Blue)
```

```
readRef : Ref a -> S Read a
```



```
six : Ref Bool -> S Read Color
six x
  = box (case readRef x of
          True  -> Red
          False -> Blue)
```

```
readRef : Ref a -> S Read a
```

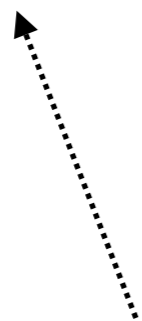
```
six : Ref Bool -> S Read Color
six x
  = case readRef x of
      True  -> Red
      False -> Blue
```

```
readRef : Ref a -> S Read a
```

`add (readRef x) (readRef y)`

`add : Nat -> Nat -> Nat`

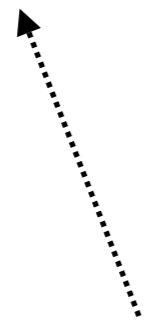
add (readRef x) (readRef y)



S Read Nat ; Pure

add : Nat -> Nat -> Nat

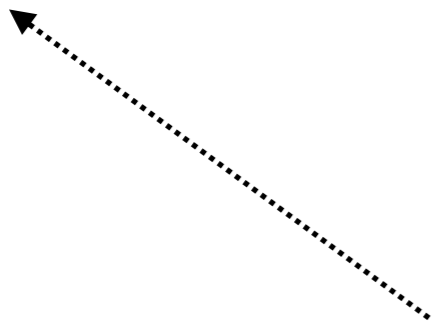
add (run readRef x) (readRef y)



Nat ; Read

add : Nat -> Nat -> Nat

```
add (run readRef x) (run readRef y)
```



```
Nat ; Read
```

```
add : Nat -> Nat -> Nat
```

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau ; \text{Pure}}$$

$$\frac{\Gamma, x:\tau_1 \vdash M \leq \tau_2 ; \text{Pure}}{\Gamma \vdash (\lambda x. M) \leq \tau_1 \rightarrow \tau_2 ; \text{Pure}}$$

$$\Gamma \vdash M_1 \Rightarrow \tau_1 \rightarrow \tau_2 ; \sigma_1 \quad \Gamma \vdash M_2 \leq \tau_1 ; \sigma_2$$

$$\Gamma \vdash M_1 M_2 \Rightarrow \tau_2 ; \sigma_1 + \sigma_2$$

$$\Gamma \vdash M \Rightarrow \tau_1 ; \sigma \quad \tau_1 \sqsubseteq \tau_2$$

$$\Gamma \vdash M \leq \tau_2 ; \sigma$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau ; \text{Pure}}$$

$$\frac{\Gamma, x:\tau_1 \vdash M \leq \tau_2 ; \text{Pure}}{\Gamma \vdash (\lambda x. M) \leq \tau_1 \rightarrow \tau_2 ; \text{Pure}}$$

$$\Gamma \vdash M_1 \Rightarrow \tau_1 \rightarrow \tau_2 ; \sigma_1 \quad \Gamma \vdash M_2 \leq \tau_1 ; \sigma_2$$

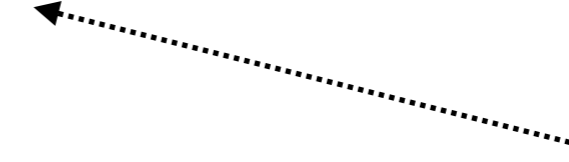
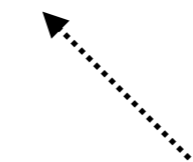
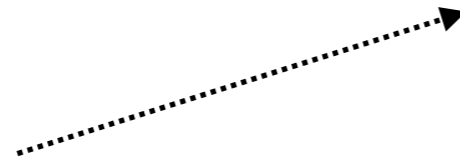
$$\Gamma \vdash M_1 M_2 \Rightarrow \tau_2 ; \sigma_1 + \sigma_2$$

$$\Gamma \vdash M \Rightarrow S \sigma_1 \tau_1 ; \sigma_2$$

$$\Gamma \vdash M \leq \tau_1 ; \sigma_1 + \sigma_2$$

Active Contexts

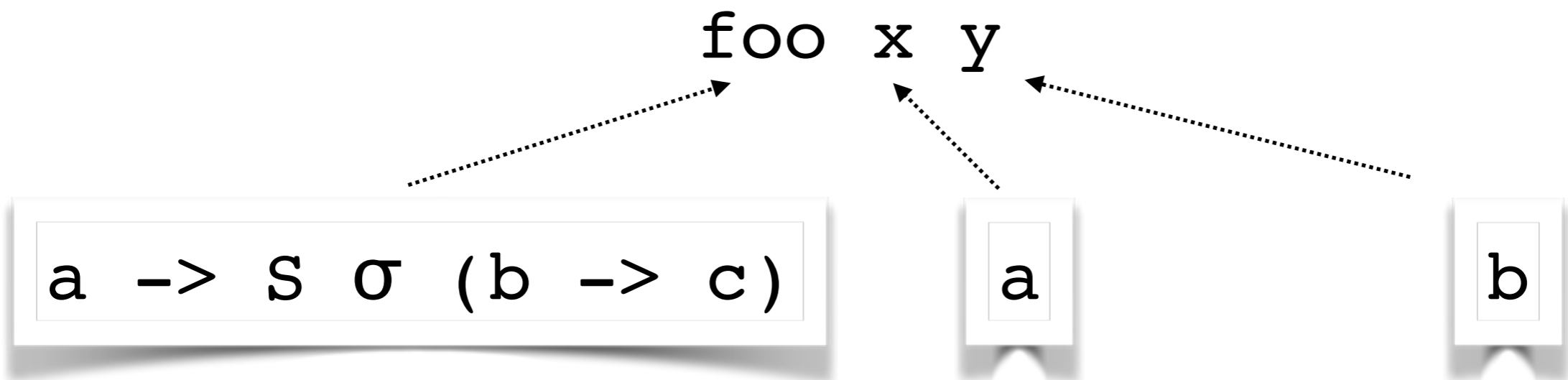
foo x y



a -> S σ (b -> c)

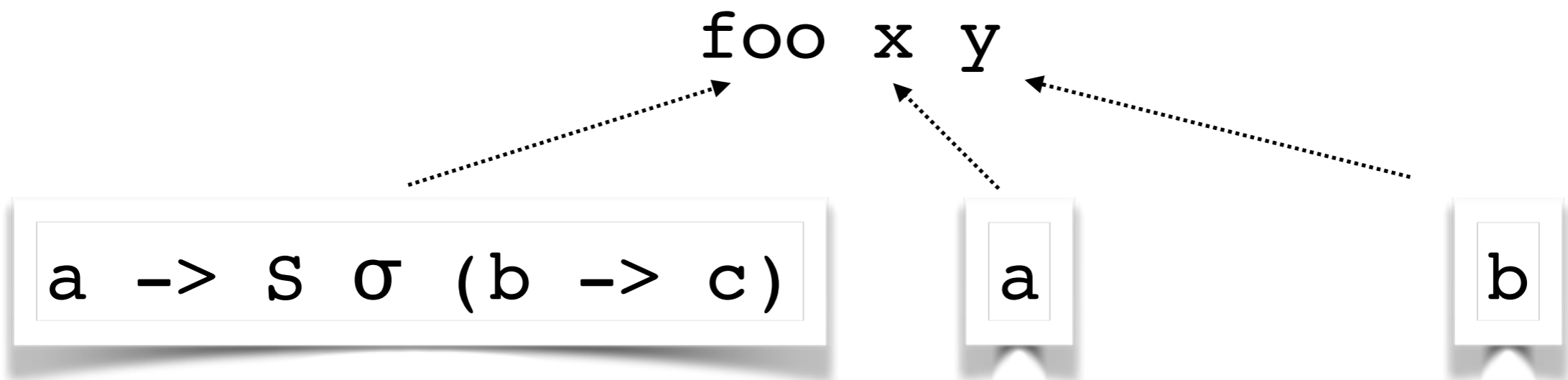
a

b



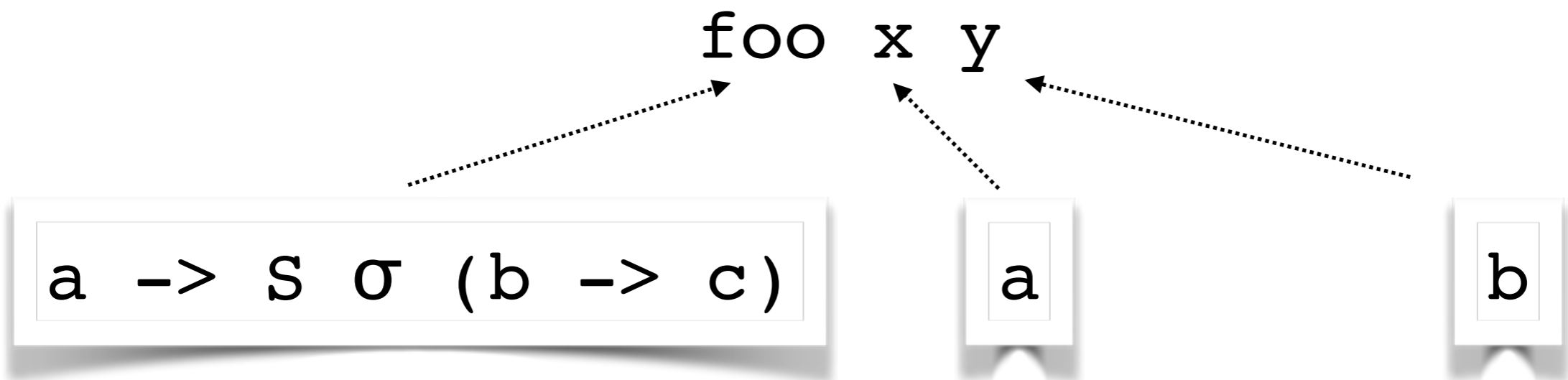
$$\begin{array}{l} \Gamma \vdash M_1 \quad \Rightarrow \tau_1 \rightarrow \tau_2 ; \sigma_1 \\ \Gamma \vdash M_2 \quad \Leftarrow \tau_1 \quad ; \sigma_2 \end{array}$$

$$\Gamma \vdash M_1 \ M_2 \quad \Rightarrow \tau_2 \quad ; \sigma_1 + \sigma_2$$

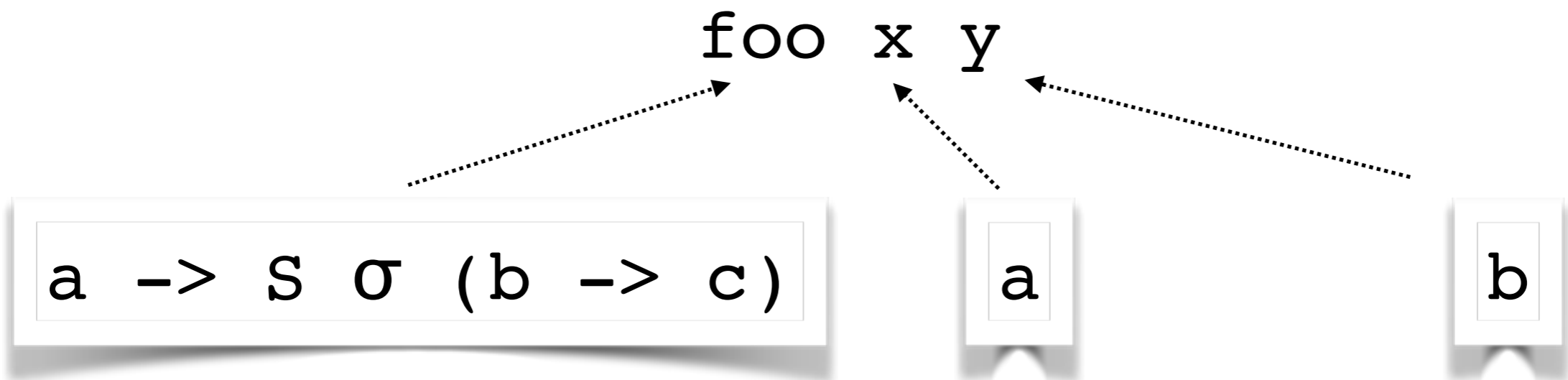


$$\frac{\begin{array}{l} \Gamma \vdash M_1 \quad /F \Rightarrow \tau_1 \rightarrow \tau_2 ; \sigma_1 \\ \Gamma \vdash M_2 \quad \leq \tau_1 \quad ; \sigma_2 \end{array}}{\Gamma \vdash M_1 M_2 \Rightarrow \tau_2 ; \sigma_1 + \sigma_2}$$

$$\Gamma \vdash M_1 M_2 \Rightarrow \tau_2 ; \sigma_1 + \sigma_2$$



$$\frac{\begin{array}{l} \Gamma \vdash M_1 \quad /F \Rightarrow \tau_1 \rightarrow \tau_2 ; \sigma_1 \\ \Gamma \vdash M_2 \quad <= \tau_1 \quad ; \sigma_2 \end{array}}{\Gamma \vdash M_1 M_2 /d \Rightarrow \tau_2 \quad ; \sigma_1 + \sigma_2}$$



$$\frac{\Gamma \vdash M_1 \quad /F \Rightarrow \tau_1 \rightarrow S \sigma_3 \tau_2 ; \sigma_1}{\Gamma \vdash M_2 \quad <= \tau_1 \quad ; \sigma_2}$$

$$\Gamma \vdash M_1 \ M_2 \ /F \Rightarrow \tau_2 \quad ; \sigma_1 + \sigma_1 + \sigma_3$$

Implemented in DDC

References

- **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
- **Joshua Dunfield and Neelakantan Krishnaswami**
Complete and Easy Bidirectional Type Checking for Higher Rank Polymorphism
Basis for type checker algorithm.

Questions?

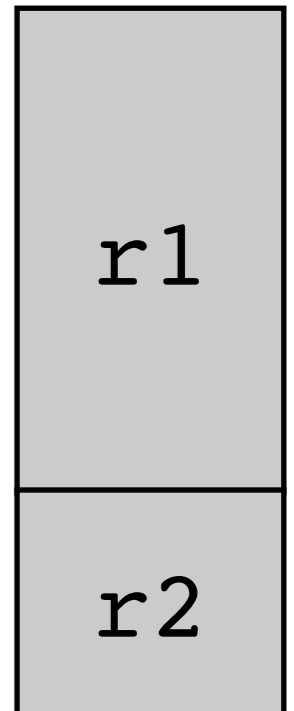
Capabilities

```
private r with {Read r; Alloc r} in  
do ref1 = run allocRef [r] 5  
...  
run readRef ref1
```

```
private r with {Read r; Alloc r} in  
do ref1 = run allocRef [r] 5  
...  
ref2 = run moar [r1] ()  
...  
run readRef ref1
```

```
moar = /\(r1 : Region). \(\x : Unit).  
  extend r1 using r2 {Write r2; Alloc r2} in  
  do ref2 = run allocRef [r2] 5  
    run writeRef [r2] ref 42  
    ref2
```

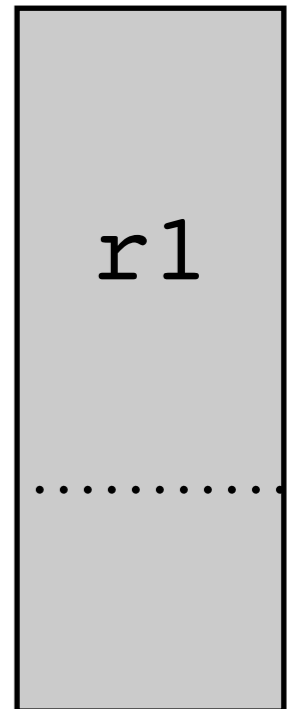
```
private r with {Read r; Alloc r} in  
do ref1 = run allocRef [r] 5  
...  
ref2 = run moar [r1] ()  
...  
run readRef ref1
```



```
moar = /\(r1 : Region). \(x : Unit).  
  extend r1 using r2 {Write r2; Alloc r2} in  
  do ref2 = run allocRef [r2] 5  
    run writeRef [r2] ref 42  
    ref2
```

Ref r2 Nat ; Alloc r2 + Write r2

```
private r with {Read r; Alloc r} in  
do ref1 = run allocRef [r] 5  
...  
ref2 = run moar [r1] ()  
...  
run readRef ref1
```



```
moar = /\(r1 : Region). \(x : Unit).
```

```
extend r1 using r2 {Write r2; Alloc r2} in  
do ref2 = run allocRef [r2] 5  
  run writeRef [r2] ref 42  
  ref2
```

Ref r1 Nat ; Alloc r1

End