# Automatically Escaping Monads

---

Ben Lippmeier
HaskellZ 2017/02/08

```
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
```

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

$$\frac{\Gamma,\ x{:}\tau_1 \vdash M : \tau_2}{\Gamma \vdash (\lambda x{:}\tau_1.\ M) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\ M_2 : \tau_2}$$

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

$$\frac{\Gamma, x{:}\tau_1 \vdash M : \tau_2 \; ; \; \mathtt{Pure}}{\Gamma \vdash (\lambda x{:}\tau_1 .\; M) : \tau_1 \to \tau_2 \; ; \; \mathtt{Pure}}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \; ; \; \sigma_1 \qquad \Gamma \vdash M_2 : \tau_1 \; ; \; \sigma_2}{\Gamma \vdash M_1\; M_2 : \tau_2 \; ; \; \sigma_1 + \sigma_2}$$

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

$$\frac{\Gamma, x{:}\tau_1 \vdash M : \tau_2 \; ; \; \mathtt{Pure}}{\Gamma \vdash (\lambda x{:}\tau_1.\ M) : \tau_1 \to \tau_2 \; ; \; \mathtt{Pure}}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \; ; \; \sigma_1 \qquad \Gamma \vdash M_2 : \tau_1 \; ; \; \sigma_2}{\Gamma \vdash M_1\ M_2 : \tau_2 \; ; \; \sigma_1 + \sigma_2}$$

$$\frac{\Gamma \vdash M : \mathtt{S}\ \sigma_1\ \tau \; ; \; \sigma_2 \qquad \Gamma\ \mathtt{supports}\ \sigma_1}{\Gamma \vdash \mathtt{run}\ M : \tau \; ; \; \sigma_1 + \sigma_2}$$

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

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

Latent effects = Bad

$$\mathtt{f} : \tau_1 \rightarrow \mathtt{S}\ \mathtt{e}\ \tau_2$$

Effect indexed computation types = Good

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

Latent effects = Bad

$$\mathtt{f} : \tau_1 \rightarrow \mathtt{S}\ \mathtt{e}\ \tau_2$$

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)
```

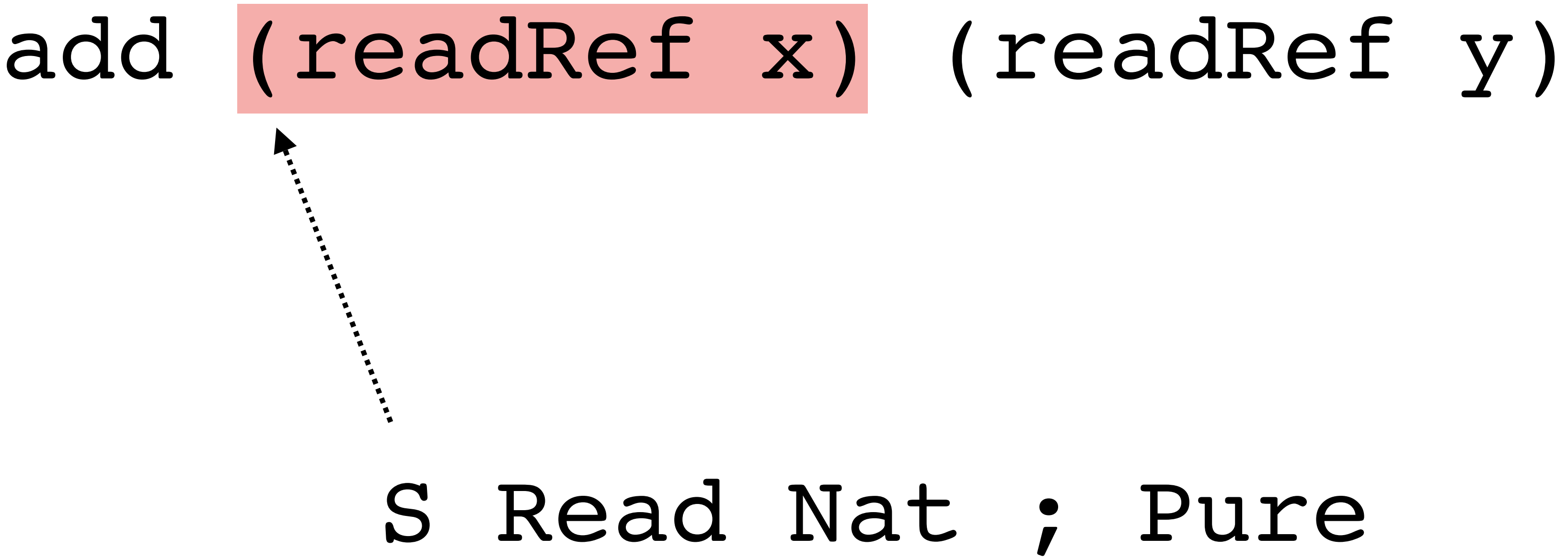
```
readRef : Ref a -> S Read a
```

```
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
```

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

$$\frac{\Gamma, x{:}\tau_1 \vdash M \Leftarrow \tau_2 \;;\; \mathrm{Pure}}{\Gamma \vdash (\lambda x.\ M) \Leftarrow \tau_1 \to \tau_2 \;;\; \mathrm{Pure}}$$

$$\frac{\Gamma \vdash M_1 \Rightarrow \tau_1 \to \tau_2 \;;\; \sigma_1 \qquad \Gamma \vdash M_2 \Leftarrow \tau_1 \;;\; \sigma_2}{\Gamma \vdash M_1\ M_2 \Rightarrow \tau_2 \;;\; \sigma_1 + \sigma_2}$$

$$\frac{\Gamma \vdash M \Rightarrow \tau_1 \;;\; \sigma \qquad \tau_1 \sqsubseteq \tau_2}{\Gamma \vdash M \Leftarrow \tau_2 \;;\; \sigma}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x => \tau \ ; \ \mathtt{Pure}}$$

$$\frac{\Gamma, \ x{:}\tau_1 \vdash M <= \tau_2 \ ; \ \mathtt{Pure}}{\Gamma \vdash (\lambda x.\ M) <= \tau_1 \to \tau_2 \ ; \ \mathtt{Pure}}$$

$$\frac{\Gamma \vdash M_1 => \tau_1 \to \tau_2 \ ; \ \sigma_1 \qquad \Gamma \vdash M_2 <= \tau_1 \ ; \ \sigma_2}{\Gamma \vdash M_1\ M_2 => \tau_2 \ ; \ \sigma_1 + \sigma_2}$$

$$\frac{\Gamma \vdash M => S\ \sigma_1\ \tau_1 \ ; \ \sigma_2}{\Gamma \vdash M <= \tau_1 \ ; \ \sigma_1 + \sigma_2}$$

# Show Desugared Code

# Regions and Capabilities

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

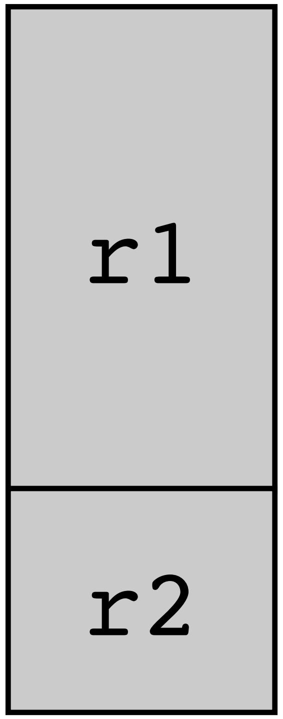
```
private r with {Read r; Alloc r} in
do ref1 = run allocRef [r] 5
   …
   run readRef ref1
```

```
allocRef : a -> S (Alloc r) (Ref r a)

readRef  : Ref r a -> S (Read r) a

writeRef : Ref r a -> a -> S (Write r) a
```

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

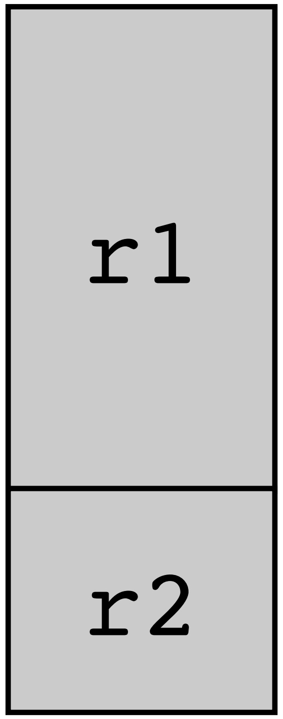


r1

r2

```
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
```

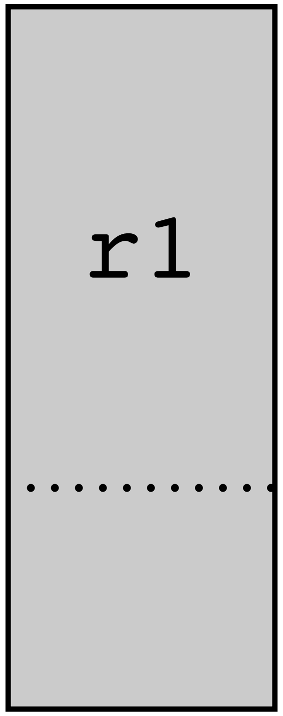
```
private r with {Read r; Alloc r} in
do ref1 = run allocRef [r] 5
   …
   ref2 = run moar [r1] ()
   …
   run readRef ref1
```



```
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
```

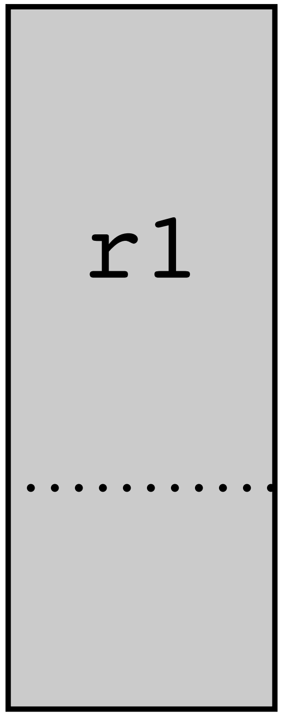


```
moar = /\(r1 : Region). \(x : Unit).
  extend r1 using r2 {Write r2; Alloc r2} in
  do ref2 = run allocRef [r2] 5
     run writeRef [r2] ref 42
     ref2
```

```
Ref r1 Nat ; Alloc r1
```

# **extend** ~ safeFreezeArray

```
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`

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”

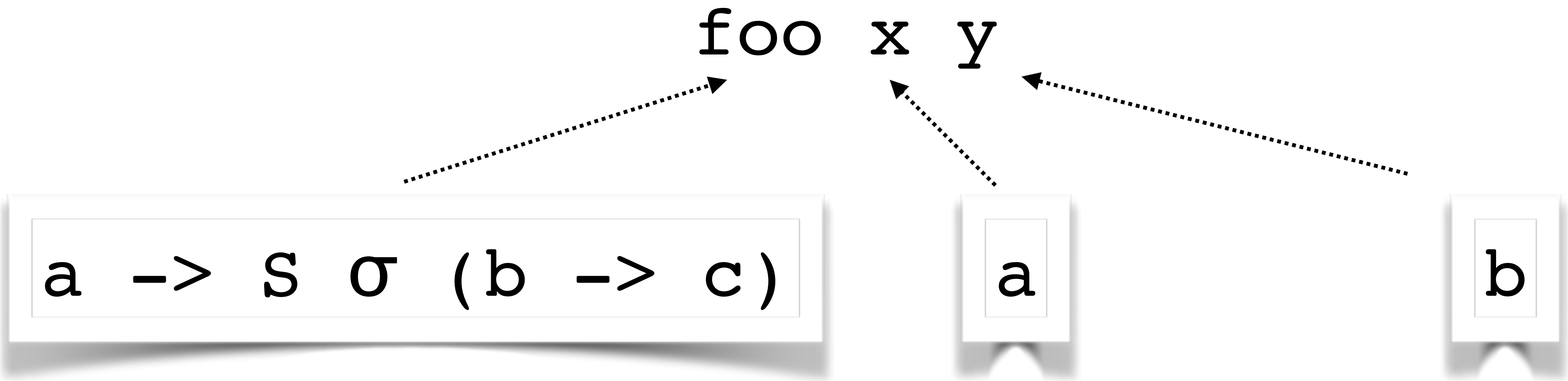
- **Paul Levy**
  *Call by Push Value*
  Has reification and reflection -like operators

- **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?

# Active Contexts

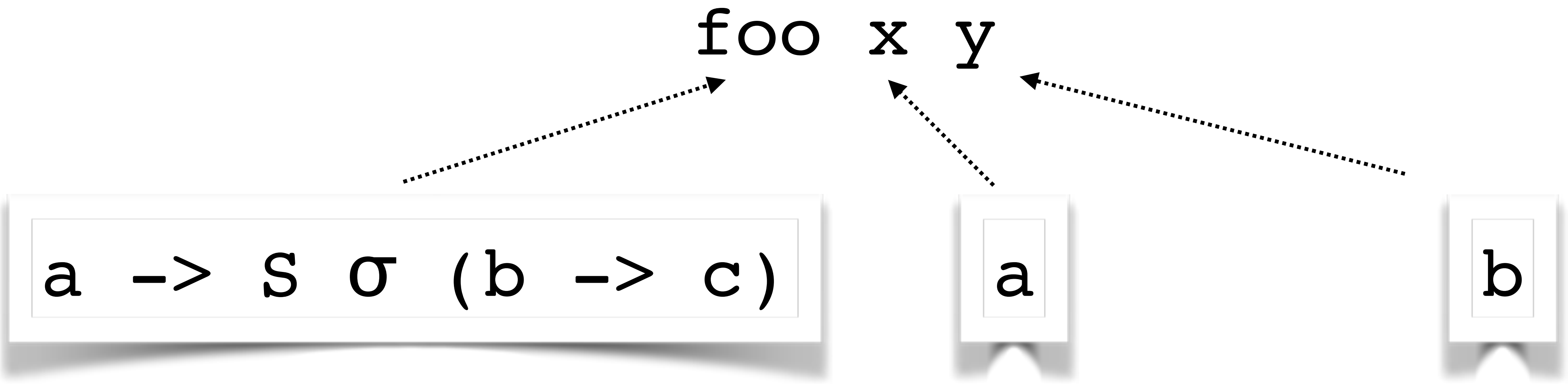


`foo x y`

`a -> S σ (b -> c)`

`a`

`b`



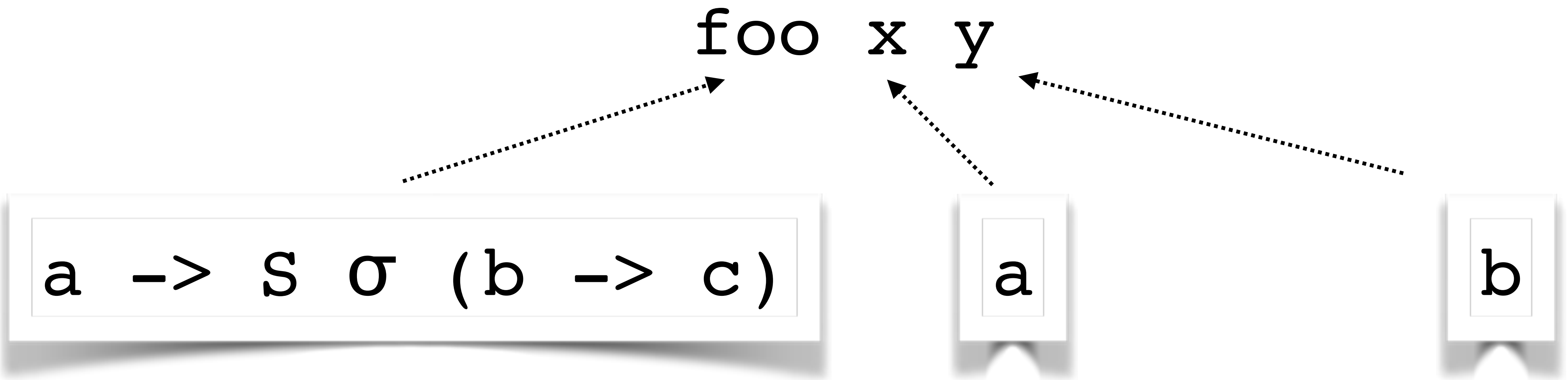
`foo x y`

`a -> S σ (b -> c)`

`a`

`b`

$$\frac{\begin{array}{llll}\Gamma \vdash M_1 & \texttt{=>}\ \tau_1 \to \tau_2 & ; & \sigma_1 \\ \Gamma \vdash M_2 & \texttt{<=}\ \tau_1 & ; & \sigma_2\end{array}}{\begin{array}{llll}\Gamma \vdash M_1\ M_2 & \texttt{=>}\ \tau_2 & ; & \sigma_1 + \sigma_2\end{array}}$$



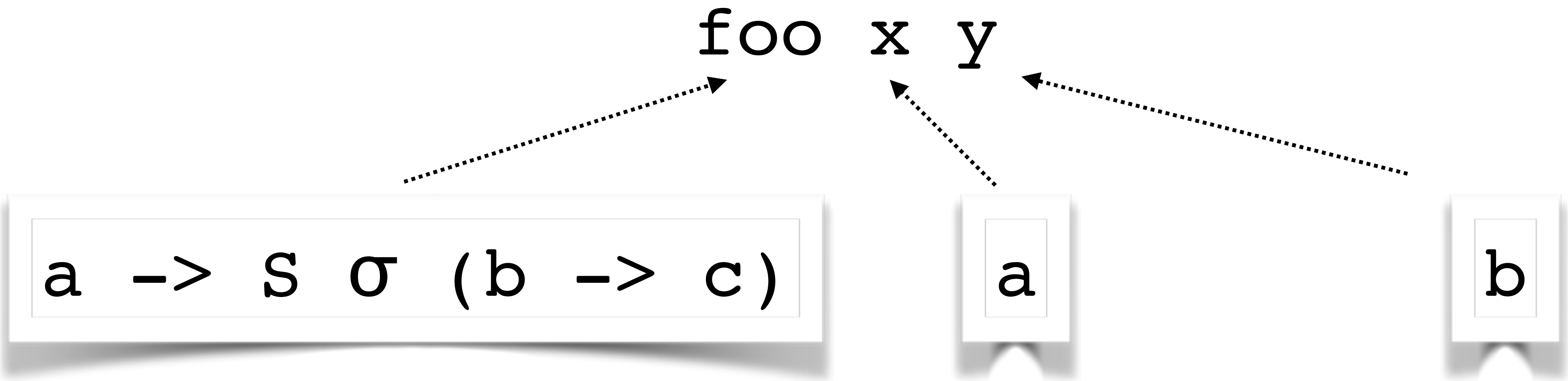
`foo x y`

`a -> S σ (b -> c)`

`a`

`b`

$$\frac{\begin{array}{lllll}\Gamma \vdash M_1 & /F & \mathtt{=>} & \tau_1 \to \tau_2 & ;\ \sigma_1 \\ \Gamma \vdash M_2 & & \mathtt{<=} & \tau_1 & ;\ \sigma_2\end{array}}{\Gamma \vdash M_1\ M_2 \quad \mathtt{=>}\ \tau_2 \quad ;\ \sigma_1 + \sigma_2}$$



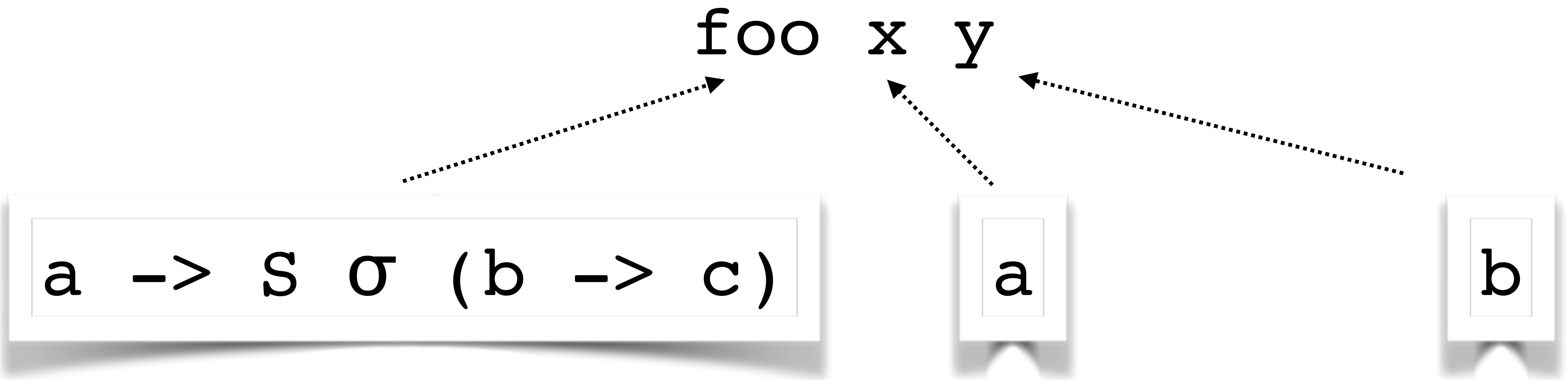
foo x y

a -> S σ (b -> c)

a

b

$$\frac{\begin{array}{llll} \Gamma \vdash M_1 & /F \Rightarrow & \tau_1 \to \tau_2 & ;\ \sigma_1 \\ \Gamma \vdash M_2 & \quad\Leftarrow & \tau_1 & ;\ \sigma_2 \end{array}}{\Gamma \vdash M_1\ M_2\ /d \Rightarrow \tau_2 \qquad ;\ \sigma_1 + \sigma_2}$$



`foo x y`

`a -> S σ (b -> c)`

`a`

`b`

$$\frac{\begin{array}{l}\Gamma \vdash M_1 \;/F \Rightarrow \tau_1 \to S\ \sigma_3\ \tau_2 \;;\; \sigma_1 \\ \Gamma \vdash M_2 \Leftarrow \tau_1 \;;\; \sigma_2\end{array}}{\Gamma \vdash M_1\ M_2 \;/F \Rightarrow \tau_2 \;;\; \sigma_1 + \sigma_1 + \sigma_3}$$