Implementing Local State with Global State

“Make Equations Great Again”

Tom Schrijvers — KU Leuven
with Shin-Chen Mu and Koen Pauwels

Overview

Make Equations Great Again

$$A \,!\, \Delta / \mathcal{E}$$

value effects equations

Make Equations Great Again

$$h : A \,!\, \Delta / \mathscr{E} \Rightarrow B \,!\, \Delta' / \mathscr{E}'$$

This Paper

$$h : A\,!\Delta/\mathcal{E}_L \Rightarrow A\,!\Delta/\mathcal{E}_G$$

where

$$\Delta = \{get, put, \|, fail\}$$

A: State

State

Interface

```
put :: s → m ()
get :: m s
```

Laws

```
get >>= \s → get >>= f s = get (\s → f s s)
             get >>= put = return ()
            put s >> get = put s (return s)
        put s1 >> put s2 = put s2
```

Standard Implementation

```
m a = s → (a, s)
```

Monad Interface

```
return x = \s → (x,s)
p >>= q  = \s → let (x,s') = p s
                 in q x s'
```

State Interface

```
put s' = \s → ((),s')
get    = \s → (s,s)
```

Bind for Free

State

Interface

```
put :: s → m ()
get :: m s
```

State

Interface

```
put :: s → m ()
get :: m s
```

Continuation-based Interface

```
putk :: s → m a → m a
getk :: (s → m a) → m a
```

State

Interface

```
put  :: s → m ()
get  :: m s
```

Continuation-based Interface

```
putk :: s → m a → m a
getk :: (s → m a) → m a

putk s p = put s >> p
getk q   = get >>= q
```

State

Interface

```
put :: s → m ()
get :: m s

put s = putk s (return ())
get   = getk return
```

Continuation-based Interface

```
putk :: s → m a → m a
getk :: (s → m a) → m a

putk s p = put s >> p
getk q   = get >>= q
```

Adapted Laws

```
putk :: s → m a → m a
getk :: (s → m a) → m a
```

State Laws

```
getk (\s1 → getk (\s2 → z s1 s2)) =  getk (\s → z s s)
                getk (\s → putk s z) =  z
                      putk s (getk z) =  putk s (z s)
                  putk s1 (putk s2 z) =  putk s2 z
```

Algebraicity

```
putk s p >>= q = putk s (p >>= q)
  getk p >>= q = getk (p >>= q)
```

Free Implementation

```
m a = Prog a
```

```
data Prog a = Ret a
            | PutK s (Prog a)
            | GetK (s → Prog a)
```

State Interface

```
putk = PutK
getk = GetK
```

Monad Interface

```
return x = Ret x
Ret x    >>= q = q x
PutK s p >>= q = PutK s (p >>= q)
GetK p   >>= q = GetK (p >=> q)
```

Handling

```
m a = Prog a
```

```
data Prog a = Ret a
            | PutK s (Prog a)
            | GetK (s → Prog a)
```

```
run :: Prog a → D a
run (Ret x)     = ret x
run (PutK s p)  = putk s (run p)
run (Get p)     = getk (run . p)
```

Semantic Domain

```
ret  :: a → D a
putk :: s → D a → D a
getk :: (s → D a) → D a
```

```
run :: Prog a → D a
run (Ret x)      = ret x
run (PutK s p)   = putk s (run p)
run (Get p)      = getk (run . p)
```

No Bind!

Equations?

```
m a = Prog a
```

```
data Prog a = Ret a
            | PutK s (Prog a)
            | GetK (s → Prog a)
```

E.g.

```
getk (\s → putk s z) = z
```

Equations?

```
m a = Prog a
```

```
data Prog a = Ret a
            | PutK s (Prog a)
            | GetK (s → Prog a)
```

E.g.

```
getk (\s → putk s z) = z
```

$$\Longleftrightarrow$$

```
GetK (\s → Putk s z) = z
```

Equations?

```
m a = Prog a
```

```
data Prog a = Ret a
            | PutK s (Prog a)
            | GetK (s → Prog a)
```

E.g.

```
getk (\s → putk s z) = z
```

$$\iff$$

```
GetK (\s → Putk s z) = z
```

Law obviously does not hold!

Contextual Equivalence

$$p =_{ctx} q \equiv (\forall C \,.\, run\ C[p] = run\ C[q])$$

Laws Revisited

```
putk :: s → m a → m a
getk :: (s → m a) → m a
```

State Laws

Domain Laws

```
getk (\s1 → getk (\s2 → z s1 s2))  =ctx  getk (\s → z s s)
                getk (\s → putk s z)  =ctx  z
                      putk s (getk z)  =ctx  putk s (z s)
                  putk s1 (putk s2 z)  =ctx  putk s2 z
```

Algebraicity

For free

```
putk s p ≫= q  =ctx  putk s (p ≫= q)
  getk p ≫= q  =ctx  getk (p ≫= q)
```

Domain Laws

```
ret  :: a → D a
putk :: s → D a → D a
getk :: (s → D a) → D a
```

State Laws

```
getk (\s1 → getk (\s2 → z s1 s2)) =  getk (\s → z s s)
                getk (\s → putk s z) =  z
                     putk s (getk z) =  putk s (z s)
                  putk s1 (putk s2 z) =  putk s2 z
```

Domain Implementation

```
D a = s → (a, s)
```

```
ret x      = \s (x,s)
putk s’ p = \s → p s’
getk p     = \s → p s s
```

No Bind!

B: Non-Determinism

Non-Determinism

Interface

```
(||) :: m a → m a → m a
fail :: m a
```

Common Non-Determinism Laws

```
p || (q || r) = (p || q) || r
    fail || q = q
    p || fail = p
```

Algebraicity

```
(p || q) ≫= r = (p ≫= r) || (q ≫= r)
    fail ≫= r = fail
```

Standard Implementation

```
m a = [a]
```

Monad Interface

```
return x = [x]
p >>= q  = concatMap q p
```

Non-Determinism Interface

```
p || q = p ++ q
fail   = []
```

Refinement

Right Distributivity

```
p >>= \x → (q x || r x) =  (p >>= q) || (p >>= r)
                p >> fail =  fail
```

Consequence

```
  (p >>= (\x → return x || return x)) ||
  (q >>= (\x → return x || return x))
= (p || q) >>= (\x → return x || return x)
= (p || q) || (p || q)
```

Refinement Implementation

```
m a = Bag a
```

Non-Determinism Interface

```
p || q = Bag.union p q
fail   = Bag.empty
```

A + B

State + Non-Determinsm

What are the interaction laws?

Local State | Global State

Local State

State Copying

Backtracking

State Copying

```
putk s (p || q)
=
(putk s p) || (putk s q)
```

Backtracking

State Copying

```
putk s (p || q)
=
(putk s p) || (putk s q)
```

Backtracking

```
putk s fail
=
fail
```

Standard Implementation

```
m a = s → [(a, s)]
```

State Interface

```
putk s' p = \s → p s'
getk p    = \s → p s s
```

Non-Determinism Interface

```
p || q = \s → p s ++ q s
fail   = \s → []
```

Example

```
m a = s → [(a, s)]
```

```
p = getk (\s → putk (s+1) fail) ||
    getk (\s → putk (s+2) (return s))


> p 0
[(0,2)]
```

Refinement Implementation

```
m a = s → Bag (a, s)
```

State Interface

```
putk s' p = \s → p s'
getk p    = \s → p s s
```

Non-Determinism Interface

```
p || q = \s → Bag.union (p s) (q s)
fail   = \s → Bag.empty
```

Local State

+ Reasoning

− Performance

Global State

Left Bias

Left Bias

```
putk s (p || q)
       =
(putk s p) || q
```

Standard Implementation

```
m a = s → (Maybe (a, m a), s)
```

State Interface

```
putk s' p = \s → p s'
getk p    = \s → p s s
```

Non-Determinism Interface

```
p || q = \s → case p s of
      (Nothing,     s') → q s'
      (Just (x,p'),s') → (Just (x,p' || q),s')
fail   = \s → (Nothing, s)
```

Example

```
m a = s → (Maybe (a, m a), s)
```

```
p = getk (\s → putk (s+1) fail)         ||
    getk (\s → putk (s+2) (return s))  ||
    getk (\s → putk (s+3) fail)
```

```
> p 0
(Just (1,(\s → (Nothing,s+3)),3)
```

Global State

− Reasoning

+ Performance

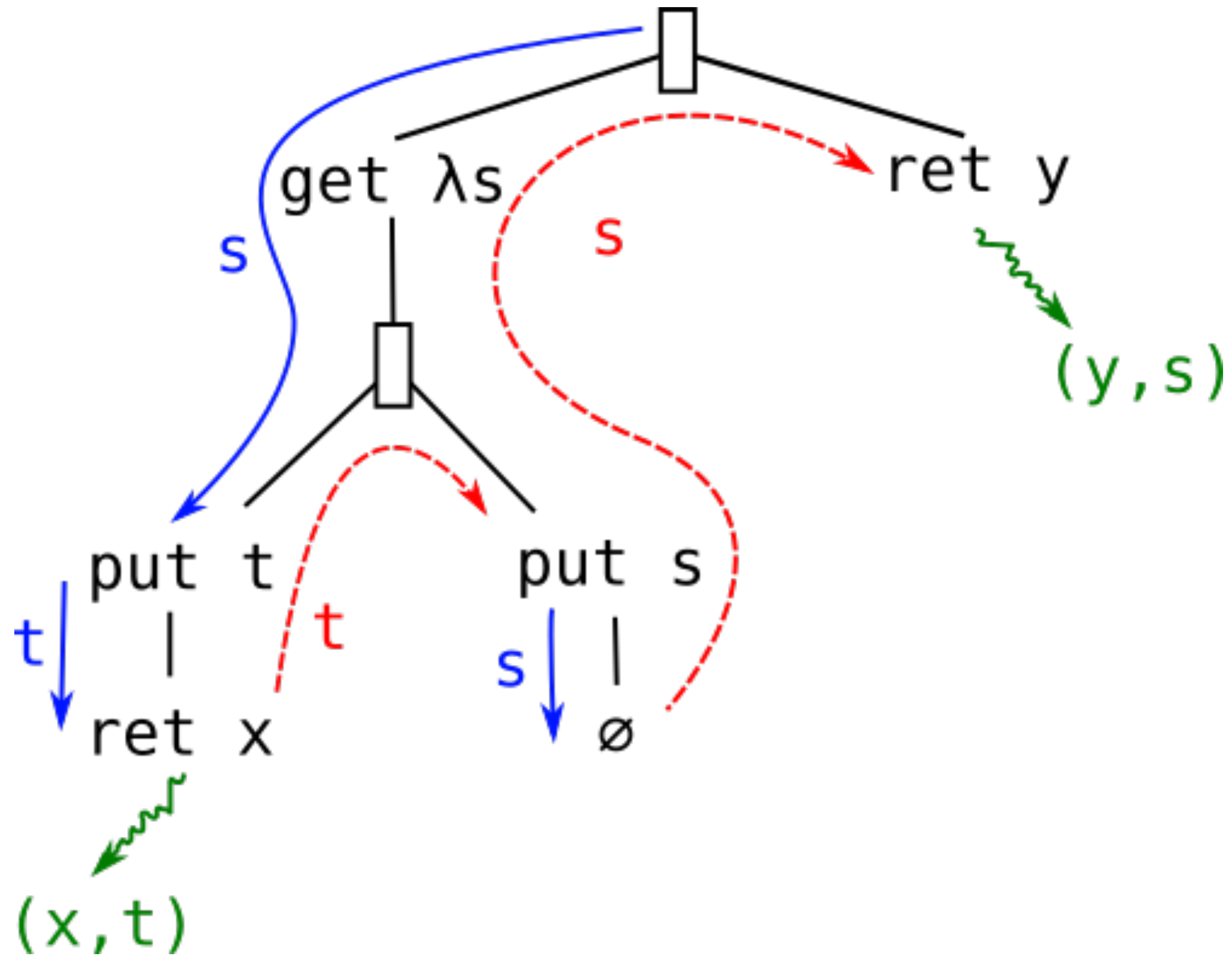
Handle Local with Global

Handling Idea

Handling Idea

```
getkᴸ p = getkᴳ p

putkᴸ s’ p =
  getkᴳ (\s →   (putkᴳ s’ p)
              || (putkᴳ s fail))
```



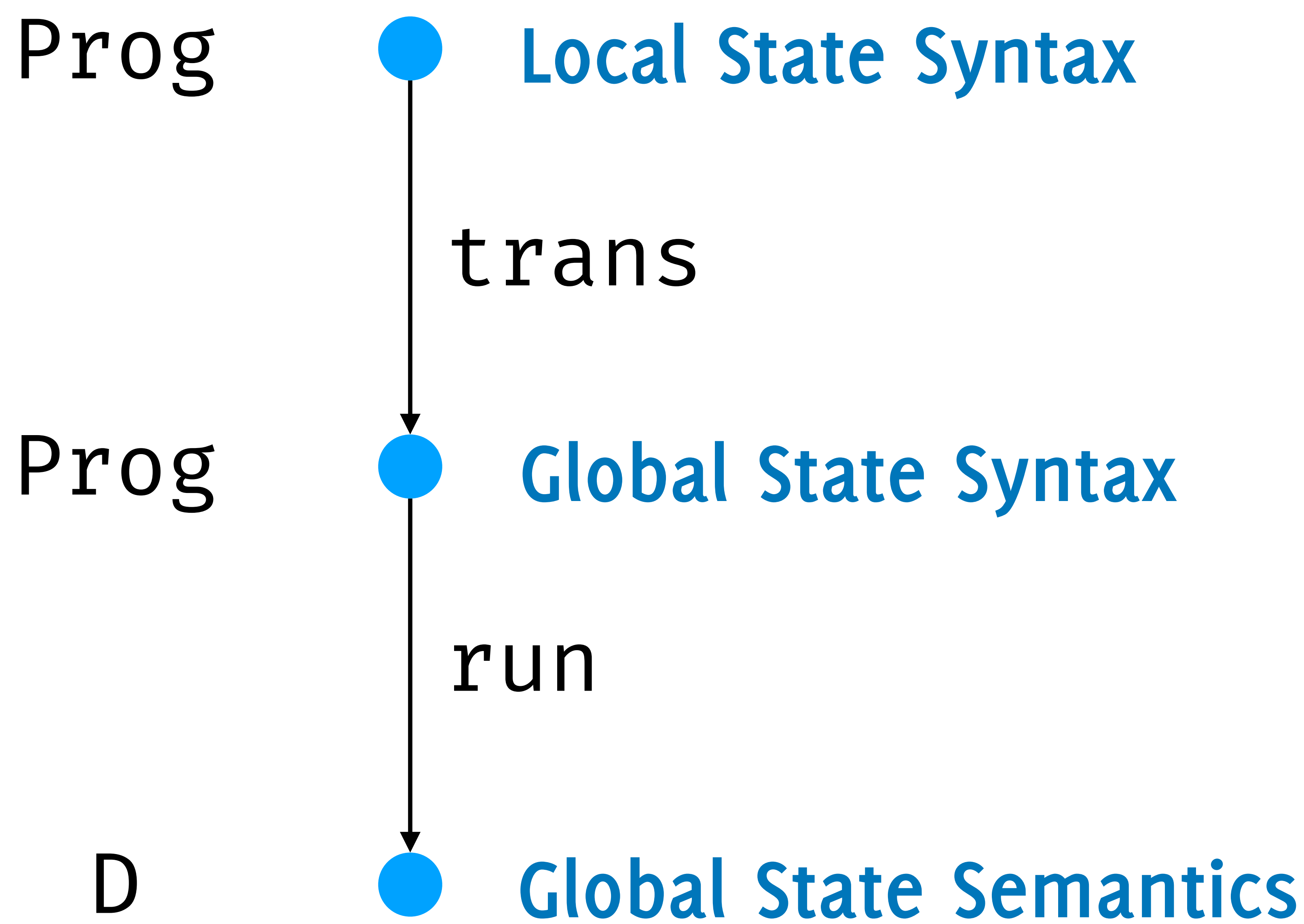
$\texttt{putk}_L\ \texttt{t (ret x) || ret y}$

Free Monad Approach

Free Monad Approach

```
data Prog a = Ret a
            | PutK s (Prog a)
            | GetK (s → Prog a)
            | Fail
            | Or (Prog a) (Prog a)
```

Free Monad Approach



Syntax to Syntax

Syntax to Syntax

```
trans :: Prog a → Prog a
trans (Ret x)      = Ret x
trans (PutK s’ p) =
  GetK (\s → Or (PutK s’ (trans p))
                (PutK s Fail))
trans (GetK p)     = GetK (trans . p)
trans Fail         = Fail
trans (Or p q)     = Or (trans p q)
```

Global Semantics

Global Semantics

```
run :: Prog a → D a
run (Ret x)      = ret x
run (PutK s p)   = putk s (run p)
run (GetK p)     = getk (run . p)
run Fail         = fail
run (Or p q)     = run p || run q
```

Contextual Equivalences

$$p =_G q \equiv (\forall C \,.\, run\, C[p] = run\, C[q])$$

$$p =_L q \equiv (\forall C \,.\, run\, (trans\, C[p]) = run\, (trans\, C[q]))$$

Handling Laws

`Prog` — Local State Syntax — Local State Law $=_L$

`trans`

`Prog` — Global State Syntax — Global State Law $=_G$

+ Non-Contextual Laws

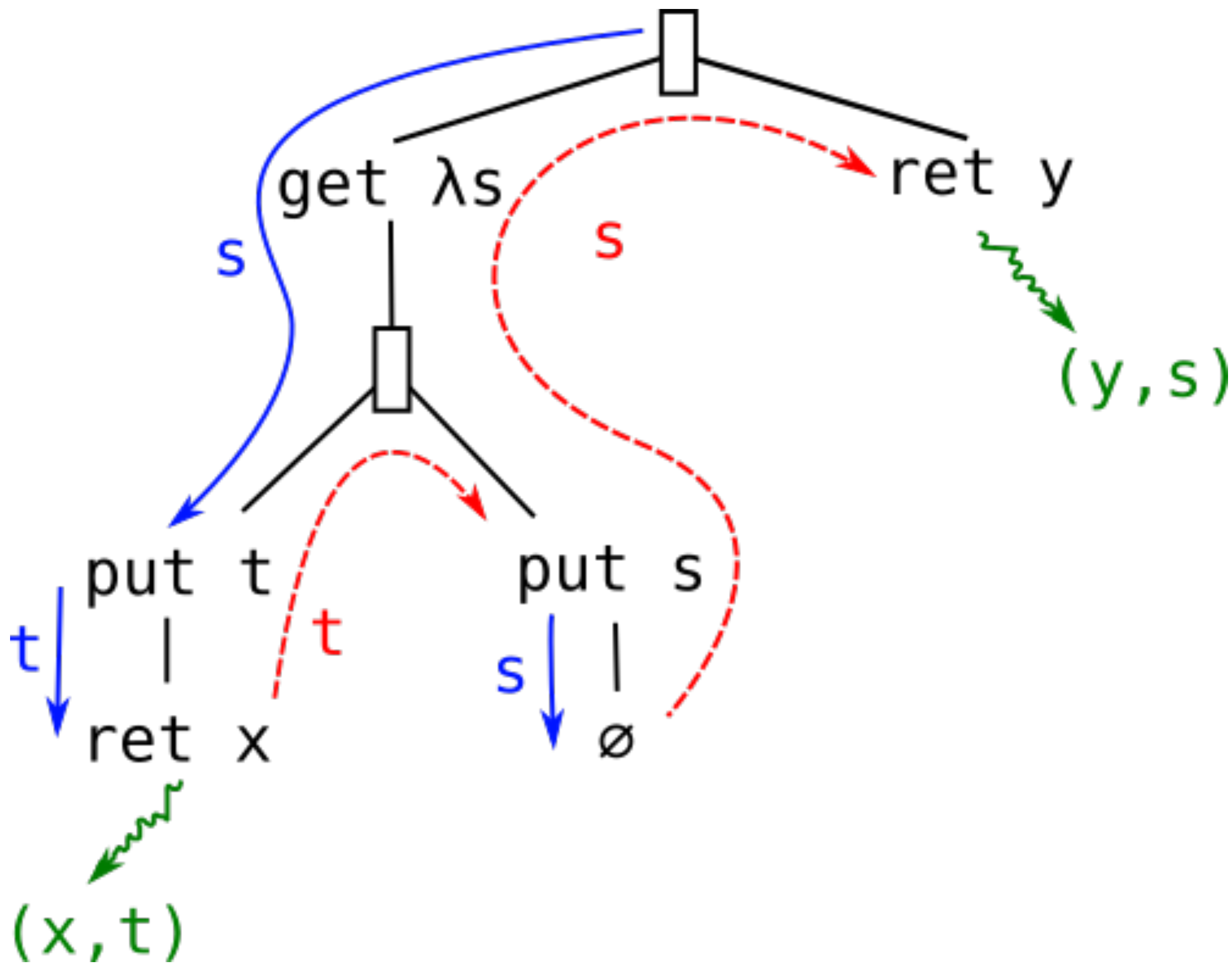
`run`

`D` — Global State Semantics — $=$

Handling Laws

Local State	Global State	
State Laws	State Laws	Non-Contextual Put Distributivity
Non-Determinism Laws	Non-Determinism Laws	
Local State Law	Global State Law	

Transfer State to Next Branch



Transfer State to Next Branch

Non-Contextual
Put Distributivity

```
run (putk s (return x || p))
=
run (putk s (return x) || putk s p)
```

Transfer State to Next Branch

Non-Contextual

Put Distributivity

```
putk s (ret x || p)
=
putk s (ret x) || putk s p
```

Standard Implementation

`m a = s → (Maybe (a, m a), s)`

Non-Contextual Put Distributivity

`putk 5 (ret x || ret y)`

= ?? =

`putk 5 (ret x) || putk 5 (ret y)`

Standard Implementation

```
m a = s → (Maybe (a, m a), s)
```

Non-Contextual
Put Distributivity

```
putk 5 (ret x || ret y)
=
\_ → (Just (x, \s → (Just (y,\s → (Nothing,s)),s)),5)
≠
\_ → (Just (x, \s → (Just (y,\s → (Nothing,s)),5)),5)
=
putk 5 (ret x) || putk 5 (ret y)
```

Standard Implementation

```
m a = s → (Maybe (a, m a), s)
```

Non-Contextual Put Distributivity

```
putk 5 (ret x || ret y)
=
\_ → (Just (x, \s → (Just (y,\s → (Nothing,s)),s)),5)
≠
\_ → (Just (x, \s → (Just (y,\s → (Nothing,s)),5)),5)
=
putk 5 (ret x) || putk 5 (ret y)
```

Broken!

Non-Standard Implementation

```
m a = s → ([(a, s)], s)
```

Non-Determinism Interface

```
p || q = \s → let (b1,s1) = p s
                  (b2,s2) = q s1
              in  (b1 ++ b2, s2)
fail   = \s → ([], s)
```

State Interface

```
put s p = \s → p s
get p   = \s → p s s
```

Not a Monad!

```
ret x = \s → ([(x,s)], s)
```

Example

```
m a = s → ([(a, s)], s)
```

```
p = getk (\s → putk (s+1) fail)      ||
    getk (\s → putk (s+2) (ret s))   ||
    getk (\s → putk (s+3) fail)
```

```
> p 0
([(1,3)],6)
```

Non-Standard Implementation

```
m a = s → ([(a, s)], s)
```

Non-Contextual

Put Distributivity

Non-Standard Implementation

```
m a = s → ([(a, s)], s)
```

Non-Contextual Put Distributivity

```
putk 5 (ret x || ret y)
=
\_ → ([(x,5),(y,5)],5)
=
\_ → (([(x,5),(y,5)],5))
=
putk 5 (ret x) || putk 5 (ret y)
```

Handling Laws

Local State

State Laws

Non-Determinism Laws

Local State Law

Right Distributivity

Global State

State Laws

Non-Determinism Laws

Global State Law

Non-Contextual Put Distributivity

Non-Contextual Commutativity

Commute Results

Non-Contextual Commutativity

```
                        getk (\s →
putk (t s) p || putk (u s) q || putk s fail)
                            =
                        getk (\s →
putk (u s) q || putk (t s) p || putk s fail)
```

Non-Standard Implementation

```
m a = s → (Bag (a, s), s)
```

Non-Determinism Interface

```
p || q = \s → let (b1,s1) = p s
                  (b2,s2) = q s1
              in  (Bag.union b1 b2, s2)
fail   = \s → (Bag.empty, s)
```

State Interface

```
put s p = \s → p s
get p   = \s → p s s
```

Not a Monad!

```
ret x = \s → (Bag.singleton (x,s), s)
```

Look Ma, No Copy

Best of Both Worlds

- Reasoning
- Performance

Modify

Standard Definition

```
modifyk f p =
  getk (\s → putk (f s) p)
```

Local as Global

Assume

```
g . f = id
```

Non-Standard Definition

```
modifykL' f g p =
   (modifykG f p) || (modifykG g fail)
```

Free Monad Approach

Free Monad Approach

```
data Progₘ a = Ret a
             | PutK s (Progₘ a)
             | GetK (s → Progₘ a)
             | Fail
             | Or (Progₘ a) (Progₘ a)
             | ModifyK (s → s) (s → s) (Progₘ a)
```

Standard
Syntax to Syntax

Standard
Syntax to Syntax

```
trans1 :: Progₘ a ⟶ Prog a
…
trans1 (Modifyk f g p) =
  GetK (\s →
    Or (PutK (f s) (trans1 p))
       (GetK (\t → PutK (g t) Fail)
  )
```

Efficient
Syntax to Syntax

Efficient
Syntax to Syntax

```
trans2 :: Progₘ a ⟶ Prog a
…
trans2 (Modifyk f g p) =
  GetK (\s →
    Or (PutK (f s) (trans2 p))
       (PutK s Fail)
  )
```

Correctness

Correctness

Theorem

```
run (trans1 p) = run (trans2 p)
```

Summary

Summary

★ We simulate local state
with global state…

★ …that satisfies 2 non-contextual laws…

★ …which are satisfied by a non-monadic implementation.

★ Free Monad/Effect Handler approach instrumental in **linking denotational and syntactic approaches** to meta-theory

Thank You!

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Prog}\ a\ \textbf{where} \\
\quad \begin{array}{ll}
\mathsf{Return} & :: a \to \mathsf{Prog}\ a \\
\emptyset & :: \mathsf{Prog}\ a \\
(\talloblong) & :: \mathsf{Prog}\ a \to \mathsf{Prog}\ a \to \mathsf{Prog}\ a \\
\mathsf{Get} & :: (\mathsf{S} \to \mathsf{Prog}\ a) \to \mathsf{Prog}\ a \\
\mathsf{Put} & :: \mathsf{S} \to \mathsf{Prog}\ a \to \mathsf{Prog}\ a
\end{array}
\end{array}
$$

(a)

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Env}\ (l :: [\,*\,])\ \textbf{where} \\
\quad \mathsf{Nil} :: \mathsf{Env}\ `[] \\
\quad \mathsf{Cons} :: a \to \mathsf{Env}\ l \to \mathsf{Env}\ (a : l) \\
\textbf{type}\ \mathsf{OProg}\ e\ a = \mathsf{Env}\ e \to \mathsf{Prog}\ a
\end{array}
$$

(b)

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Ctx}\ e_1\ a\ e_2\ b\ \textbf{where} \\
\quad \begin{array}{ll}
\square & :: \mathsf{Ctx}\ e\ a\ e\ a \\
\mathsf{COr1} & :: \mathsf{Ctx}\ e_1\ a\ e_2\ b \to \mathsf{OProg}\ e_2\ b \\
 & \to \mathsf{Ctx}\ e_1\ a\ e_2\ b \\
\mathsf{COr2} & :: \mathsf{OProg}\ e_2\ b \to \mathsf{Ctx}\ e_1\ a\ e_2\ b \\
 & \to \mathsf{Ctx}\ e_1\ a\ e_2\ b \\
\mathsf{CPut} & :: (\mathsf{Env}\ e_2 \to \mathsf{S}) \to \mathsf{Ctx}\ e_1\ a\ e_2\ b \\
 & \to \mathsf{Ctx}\ e_1\ a\ e_2\ b \\
\mathsf{CGet} & :: (\mathsf{S} \to \mathsf{Bool}) \to \mathsf{Ctx}\ e_1\ a\ (\mathsf{S} : e_2)\ b \\
 & \to (\mathsf{S} \to \mathsf{OProg}\ e_2\ b) \to \mathsf{Ctx}\ e_1\ a\ e_2\ b \\
\mathsf{CBind1} & :: \mathsf{Ctx}\ e_1\ a\ e_2\ b \to (b \to \mathsf{OProg}\ e_2\ c) \\
 & \to \mathsf{Ctx}\ e_1\ a\ e_2\ c \\
\mathsf{CBind2} & :: \mathsf{OProg}\ e_2\ a \to \mathsf{Ctx}\ e_1\ b\ (a : e_2)\ c \\
 & \to \mathsf{Ctx}\ e_1\ b\ e_2\ c
\end{array}
\end{array}
$$

(c)

$$
\begin{array}{rl}
\mathit{run} & :: \mathsf{Prog}\ a \to \mathsf{Dom}\ a \\
\langle \mathit{ret} \rangle & :: a \to \mathsf{Dom}\ a \\
\langle \varnothing \rangle & :: \mathsf{Dom}\ a \\
\langle \talloblong \rangle & :: \mathsf{Dom}\ a \to \mathsf{Dom}\ a \to \mathsf{Dom}\ a \\
\langle \mathit{get} \rangle & :: (\mathsf{S} \to \mathsf{Dom}\ a) \to \mathsf{Dom}\ a \\
\langle \mathit{put} \rangle & :: \mathsf{S} \to \mathsf{Dom}\ a \to \mathsf{Dom}\ a
\end{array}
$$

(d)