# TRANSFORMERS

## HANDLERS IN DISGUISE


**Nicolas Wu**
University of Bristol
nicolas.wu@bristol.ac.uk
with Tom Schrijvers


Tallinn, 26 November 2015

University of BRISTOL


PROTECT
DESTROY
TRANSFORMERS

# EFFECT HANDLERS

# Effect Handlers

## Syntax

scaffolding

```
data Free f a
  = Var a
  | Con (f (Free f a))
```

structure

```
data StateF s k
  = GetF   (s  → k)
  | PutF s (() → k)

instance Functor (StateF s) where
  fmap f (GetF k)   = GetF (f . k)
  fmap f (PutF s k) = PutF s (f . k)
```

## Semantics

carrier

```
s → (a, s)
```

handler

```
handleState :: Free (StateF s) a
            → (s → (a, s))
handleState =
  handle algState genState
```

generator

```
genState :: a → (s → (a, s))
```

algebra

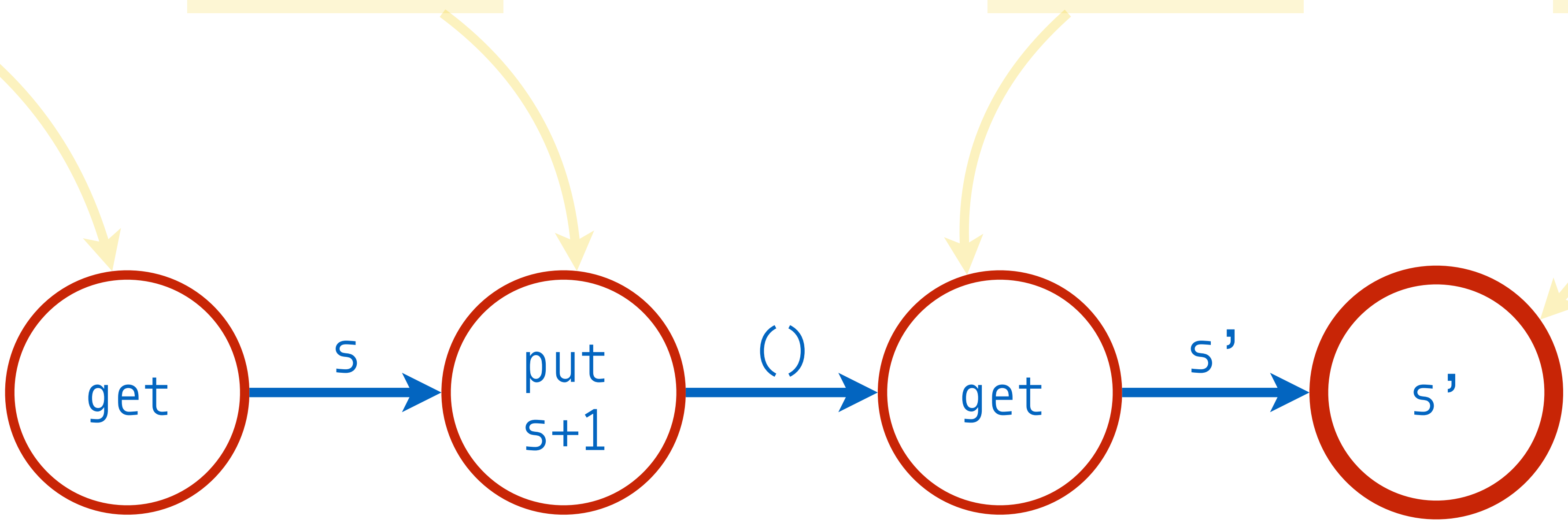
```
algState :: StateF s (s → (a, s))
                   → (s → (a, s))
```

# Syntax

```
data Free f a
  = Var a
  | Con (f (Free f a))
```

```
data StateF s k
  = GetF   (s  → k)
  | PutF s (() → k)
```

```
prog :: Free (StateF Int) Int
prog =
  Con (GetF (\s → Con (PutF (s + 1) (\() → Con (GetF (\s' → Var s'))))))
```

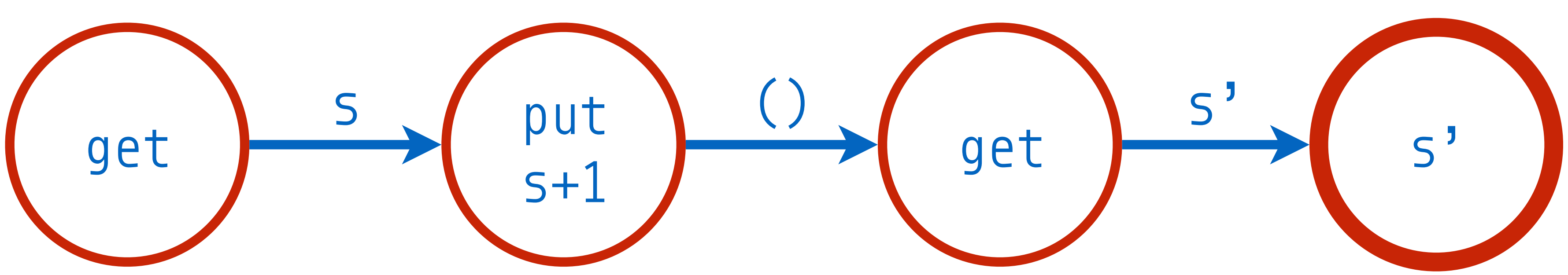


# Syntax

```
data Free f a
  = Var a
  | Con (f (Free f a))
```

```
data StateF s k
  = GetF    (s  → k)
  | PutF s (() → k)
```

```
prog :: Free (StateF Int) Int
prog = Con (GetF         (\s  →
       Con (PutF (s + 1) (\() →
       Con (GetF         (\s' →
       Var s'))))))
```

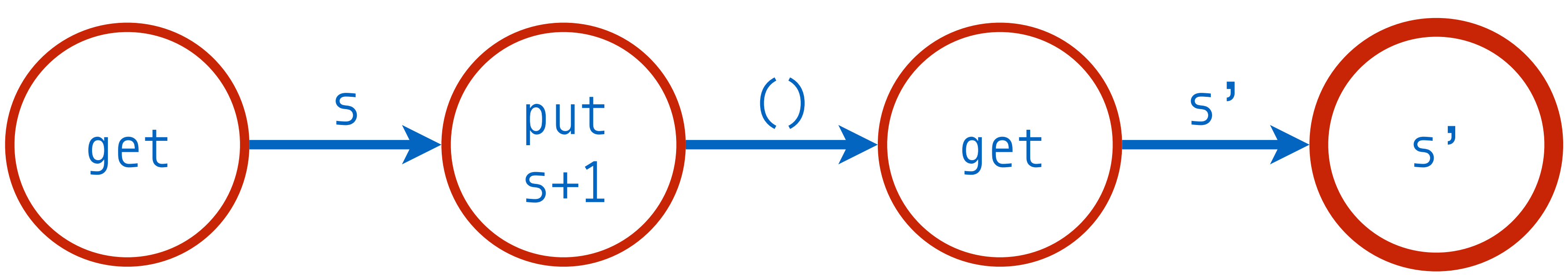


# Syntax

```
data Free f a
  = Var a
  | Con (f (Free f a))
```

```
data StateF s k
  = GetF    (s  → k)
  | PutF s (() → k)
```

```
prog :: Free (StateF Int) Int
prog = Con $ GetF          $ \s  →
       Con $ PutF (s + 1) $ \() →
       Con $ GetF          $ \s' →
       Var s'
```

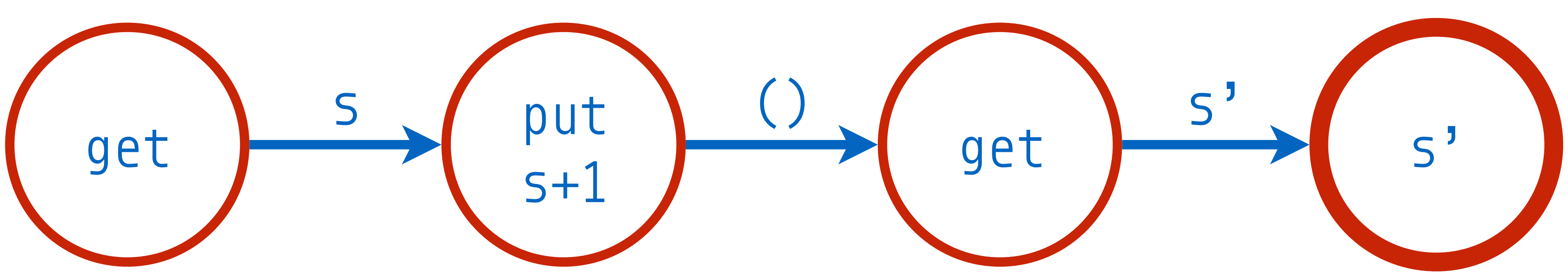


# Syntax

```
data Free f a
  = Var a
  | Con (f (Free f a))
```

```
data StateF s k
  = GetF   (s  → k)
  | PutF s (() → k)
```

```
prog :: Free (StateF Int) Int
prog = con $ GetF          $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF          $ \s' →
       var s'
  where
    con = Con
    var = Var
```

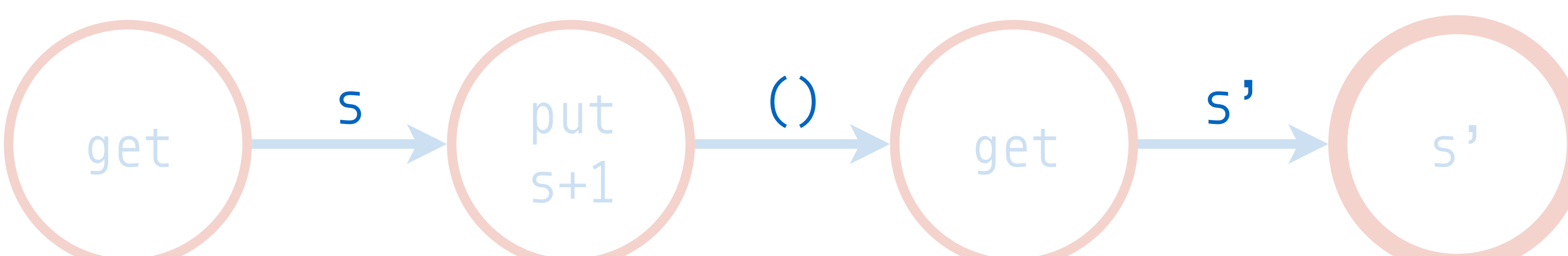


# Syntax

```
data Free f a
  = Var a
  | Con (f (Free f a))
```

```
data StateF s k
  = GetF   (s   → k)
  | PutF s (() → k)
```

```
prog :: Free (StateF Int) Int
prog = con $ GetF          $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF          $ \s' →
       var s'
  where
    con = Con
    var = Var
```



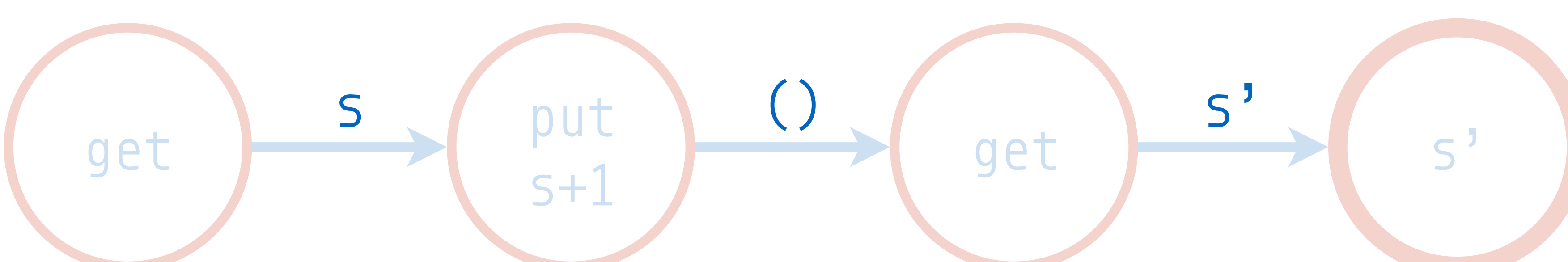
# Syntax

```
data Free f a
  = Var a
  | Con (f (Free f a))
```

```
data StateF s k
  = GetF   (s  → k)
  | PutF s (() → k)
```

```
prog :: Free (StateF Int) Int
prog = con $ GetF          $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF          $ \s' →
       var s'
  where
    con = Con
    var = Var
```

we want to give a semantics to this code



# Semantics

```
genState :: a → (s → (a, s))
genState x =  \s → (x, s)
```

```
algState :: StateF s (s → (a, s))
                     → (s → (a, s))
algState (GetF k)    = \s → k s s
algState (PutF s' k) = \s → k () s'
```

```
prog :: Int → (Int, Int)
prog = con $ GetF           $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF           $ \s' →
       var s'
  where
    con = algState
    var = genState
```

we want to give a semantics to this code

```
handle :: Functor f ⇒ (f b → b) → (a → b) → Free f a → b
handle alg gen (Var x)  = gen x
handle alg gen (Con op) = alg (fmap (handle alg gen) op)
```

# Semantics

```haskell
genState :: a → (s → (a, s))
genState x =  \s → (x, s)
```

```haskell
algState :: StateF s (s → (a, s))
                    → (s → (a, s))
algState (GetF k)    = \s → k s s
algState (PutF s' k) = \s → k () s'
```

```haskell
prog :: Int → (Int, Int)
prog = con $ GetF          $ \s  →
       con $ PutF (s + 1)  $ \() →
       con $ GetF          $ \s' →
       var s'
  where
    con = algState
    var = genState
```

we want to give a semantics to this code

to do so, the scaffolding becomes an algebra and generator and the type becomes the carrier

```haskell
handle :: Functor f ⇒ (f b → b) → (a → b) → Free f a → b
handle alg gen (Var x)  = gen x
handle alg gen (Con op) = alg (fmap (handle alg gen) op)
```

# Effect Handlers

```
prog :: Free (StateF Int) Int
prog = con $ GetF            $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF            $ \s' →
       var s'
  where
    con = Con
    var = Var
```

```
prog :: Int → (Int, Int)
prog = con $ GetF            $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF            $ \s' →
       var s'
  where
    con = algState
    var = genState
```

`handle algState genState`

the semantics is given by a fold

```
handle :: Functor f ⇒ (f b → b) → (a → b) → Free f a → b
handle alg gen (Var x)  = gen x
handle alg gen (Con op) = alg (fmap (handle alg gen) op)
```

# Effect Handlers

```
prog :: Free (StateF Int) Int
prog = con $ GetF           $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF           $ \s' →
       var s'
  where
    con = Con
    var = Var
```

```
prog :: Int → (Int, Int)
prog = con $ GetF           $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF           $ \s' →
       var s'
  where
    con = algState
    var = genState
```

handle algState genState

the semantics is given by a fold

```
handle :: Functor f ⇒ (f b → b) → (a → b) → Free f a → b
handle alg gen (Var x)  = gen x
handle alg gen (Con op) = alg (fmap (handle alg gen) op)
```

# Effect Handlers

how might we improve this?

2. wrap up the semantics

```
prog :: Free (StateF Int) Int
prog = con $ GetF          $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF          $ \s' →
       var s'
  where
    con = Con
    var = Var
```

```
prog :: Int → (Int, Int)
prog = con $ GetF          $ \s  →
       con $ PutF (s + 1) $ \() →
       con $ GetF          $ \s' →
       var s'
  where
    con = algState
    var = genState
```

1. clean up the syntax

handle algState genState

the semantics is given by a fold

```
handle :: Functor f ⇒ (f b → b) → (a → b) → Free f a → b
handle alg gen (Var x)  = gen x
handle alg gen (Con op) = alg (fmap (handle alg gen) op)
```

1. clean up the syntax

# ENTER THE MONAD

1. clean up the syntax

# Monads

monads are a standard way of encoding sequential operations

```
class Monad m where
  return :: m a
  (>>=)  :: m a → (a → m b) → m b
```

typically we use bind to say that one action must be performed before another

1. clean up the syntax

# Free Monad

```
data Free f a
  = Var a
  | Con (f (Free f a))
```

```
instance Functor f ⇒ Monad (Free f) where
  return = Var
  Var x  >>= k = k x
  Con op >>= k = Con (fmap (>>= k) op)
```

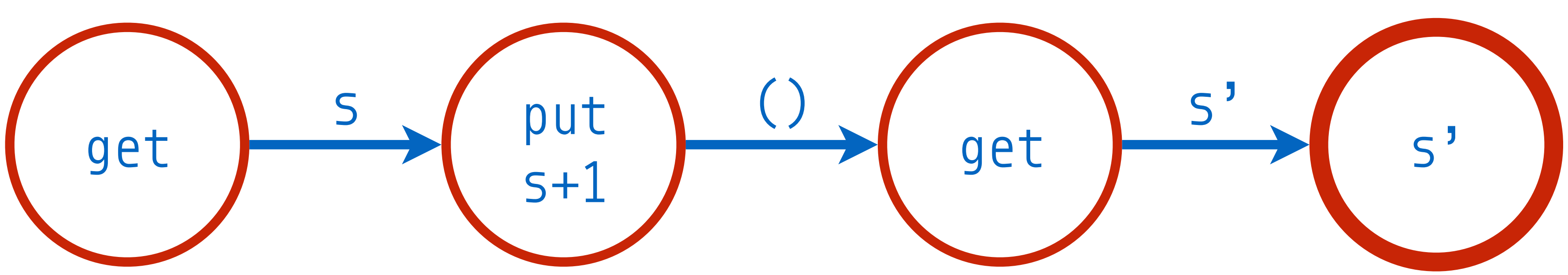
the bind for the free monad is used to graft syntax trees into variables

1. clean up the syntax

# Syntax

this is a monolithic piece of code:

```
prog :: Free (StateF Int) Int
prog = Con $ GetF         $ \s  ->
       Con $ PutF (s + 1) $ \() ->
       Con $ GetF         $ \s' ->
       Var s'
```

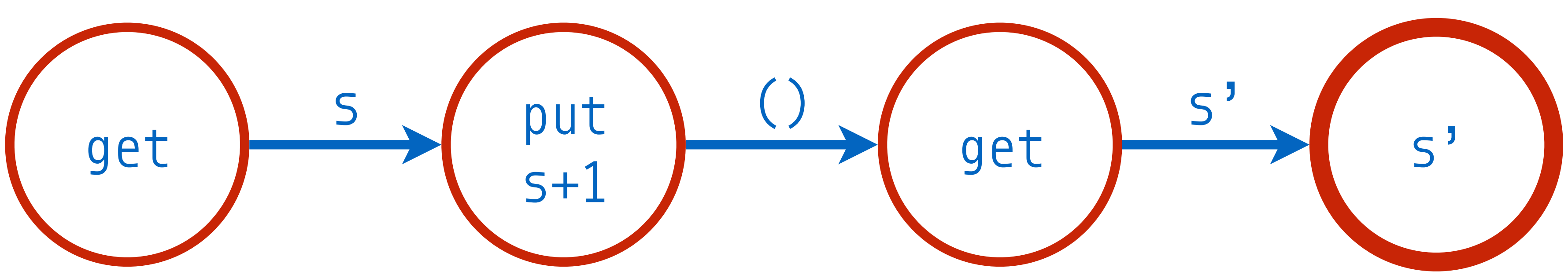


1. clean up the syntax

# Syntax

```
prog :: Free (StateF Int) Int
prog = Con (GetF Var)                >>= \s  ->
       Con (PutF (s + 1) Var) >>= \()  ->
       Con (GetF Var)                >>= \s' ->
       Var s'
```

the free monad allows us to turn
it into smaller pieces of code
that compose together to make a whole

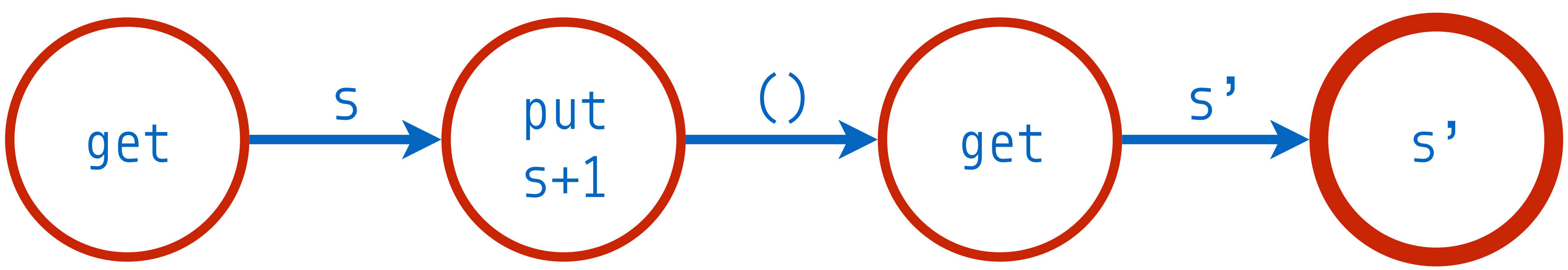


1. clean up the syntax

# Syntax

```
prog :: Free (StateF Int) Int
prog = do s  ← Con (GetF Var)
          Con (PutF (s + 1) Var)
          s' ← Con (GetF Var)
          Var s'
```

since it’s monadic, we can use Haskell’s **do** notation to make the syntax look nicer

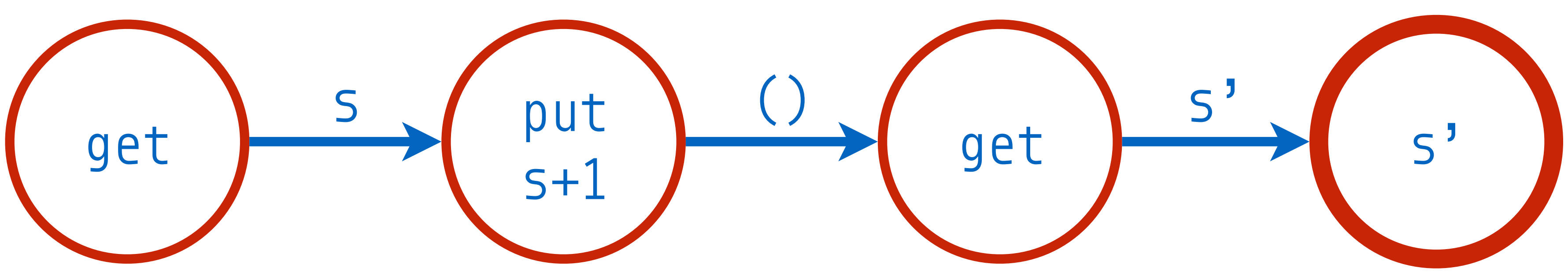


1. clean up the syntax

# Syntax

```
prog :: Free (StateF Int) Int
prog = do s  ← get
          put (s + 1)
          s' ← get
          return s'

  where
    put s = Con (PutF s Var)
    get   = Con (GetF Var)
```

finally, we can create smart constructors
to hide away some of the mess



# MONADIC SEMANTICS

2. wrap up the semantics

# State

the carrier can be wrapped in a newtype

```
newtype State s a = State { runState :: s → (a, s) }
```

we know that this happens to be a monad

```
instance Monad (State s) where
  return x        = State (\s → (x, s))
  State mx >>= f = State (\s → let (a, s') = mx s in runState (f a) s')
```

it also helps to create a specification (with laws) around the functionality this monad brings

```
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
```

```
instance MonadState s (State s) where
  get    = State (\s → (s , s ))
  put s' = State (const ((), s'))
```

2. wrap up the semantics

# State

```
newtype State s a = State { runState :: s → (a, s) }
```

the definition of a state handler becomes easy

```
genState :: a → (s → (a, s))
genState x =  \s → (x, s)

algState :: StateF s (s → (a, s)) → (s → (a, s))
algState (GetF k)     = \s → k s s
algState (PutF s' k) = \s → k () s'
```

State

```
genState :: a → State s a
genState = return

algState :: StateF s (State s a) → State s a
algState (GetF k)     = get    >>= k
algState (PutF s' k) = put s' >>= k
```

# State

## the syntax for our program is the same

### monadic style

```
prog :: State Int Int
prog = do s ← get
          put (s + 1)
          s' ← get
          return s'
```

### effect handler style

```
prog :: Free (StateF Int) Int
prog = do s ← get
          put (s + 1)
          s' ← get
          return s'
```

## we can bring these into a common framework*

```
prog :: MonadState Int m ⇒ m Int
prog = do s ← get
          put (s + 1)
          s' ← get
          return s'
```

*we've had to bend the rules since the handler put and get do not satisfy the laws

# EFFECT CLASSES

# Classy Data

The monadic specification for State is:

```
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
```

can we abstract?

the signature can be encoded with a GADT:

```
data StateS s a where
  Get :: StateS s s
  Put :: s → StateS s ()
```

Plotkin & Power call this a generic effect

and we can tie the syntax to a monadic semantics:

```
class Monad m ⇒ MonadEff f m | m → f where
  eff :: f a → m a

instance MonadEff (StateS s) (State s) where
  eff (Get)   = get
  eff (Put s) = put s
```

**Q. how does this relate to handlers?**

# GADTs vs Syntax

effect handlers use a functor:

```
data StateF s k where
  GetF ::      (s  → k) → StateF s k
  PutF :: s → (() → k) → StateF s k
```

the type class induces a GADT:

```
data StateS s a where
  Get :: StateS s s
  Put :: s → StateS s ()
```

there are some similarities between the two forms of syntax but one problem is that **StateS s** is not always functorial!

Q. can we somehow force it to have functor structure?

A. Yes, we Kan!

*left Kan extension along Id

# CoYoneda*

the CoYoneda construction adds functorial structure for free

```
data CoYoneda f r = forall a . CoYoneda (f a) (a → r)
instance Functor (CoYoneda f) where
  fmap f (CoYoneda op k) = CoYoneda op (f . k)
```

the secret is to store the outgoing continuation

now we recover constructors that are essentially the same

```
data StateF s k where
  GetF ::      (s  → k) → StateF s k
  PutF :: s → (() → k) → StateF s k

CoYoneda Get      ::      (s → k) → CoYoneda (StateS s) k
CoYoneda (Put s') :: s → (() → k) → CoYoneda (StateS s) k

instance MonadState s (Free (CoYoneda (StateS s))) where
  get    = Con (CoYoneda Get Var)
  put s' = Con (CoYoneda (Put s') Var)
```

**Q. how does this relate to handlers?**

# Monad Homomorphisms

the handler is now trivial!

```
handleCY :: MonadEff f m ⇒ Free (CoYoneda f) a → m a
handleCY = handle algCY return

algCY :: MonadEff f m ⇒ CoYoneda f (m a) → m a
algCY (CoYoneda op k) = eff op >>= k
```

in fact, what we have here is a monad homomorphism

# COMPOSITION

# Compositional Handlers

So far, we've shown how handlers relate to monads,
things get interesting when we consider handlers over composed effects

```
data (f + g) a
  = Inl (f a)
  | Inr (g a)
```

Ideally, we'd like something a bit like this:

```
Free (f + g) a → Free g b → c
```

in practice, this is too simple for an arbitrary f and g

# Compositional Handlers

for State, we need to augment the carrier significantly

```
(s → Free g (s, a))
```

```
handleState2 :: forall a s g . Functor g
             ⇒ Free (StateF s + g) a → s → Free g (s, a)
handleState2 = handle algState2 varState2
  where

    algState2 :: ((StateF s) + g) (s → Free g (s, a)) → s → Free g (s, a)
    algState2 (Inl (GetF k))    s = k s s
    algState2 (Inl (PutF s' k)) s = k () s'
    algState2 (Inr op)          s = Con (fmap ($ s) op)

    varState2 :: a → s → Free g (s, a)
    varState2 a s = return (s, a)
```

this handler generates a second tree wrapped in structure

```
handleExcState :: Free (StateF s + ExcF) a → s → Maybe (s, a)
handleExcState p s = handleExc (handleState2 p s)
```

can this be simplified?

# TRANSFORMERS

# State Transformer

hmm, this type looks familiar

```
newtype StateT s m a = StateT { runStateT :: s → m (a, s) }

instance Monad m ⇒ MonadState s (StateT s m) where
  get   = StateT (\ s → return (s,  s))
  put s = StateT (\ _ → return ((), s))
```

```
class MonadTrans t where
  lift :: Monad m ⇒ m a → t m a

instance MonadTrans (StateT s) where
  lift m = StateT $ \ s → do
    a ← m
    return (a, s)
```

```
instance Monad m ⇒ MonadEff (StateS s) (StateT s m) where
  eff (Get)   = get
  eff (Put s) = put s
```

# Semantics Transformers

```
handleT :: (MonadTrans t, MonadEff f (t (Free g)), Functor g)
        ⇒ Free (CoYoneda f + g) a → t (Free g) a
handleT (Var x)                          =  return x
handleT (Con (Inl (CoYoneda x k)))       =  eff x >>= handleT . k
handleT (Con (Inr op))                   =  join (lift (inj (fmap handleT op)))

inj :: Functor f ⇒ f a → Free f a
inj  =  Con . fmap Var
```

# Chaining Transformers

```
handleT :: (MonadTrans t, MonadEff f (t (Free g)), Functor g)
        ⇒ Free (CoYoneda f + g) a → t (Free g) a
```

how can we compose these handlers?

```
handle2  :: ( HFunctor t, MonadTrans t
            , MonadEff f (t (Free (CoYoneda g)))
            , MonadEff g m )
         ⇒ Free (CoYoneda f + CoYoneda g) a → t m a
handle2 = hmap handleCY . handleT

class HFunctor h where
  hmap :: (Functor f, Functor g) ⇒
    (forall a . f a → g a) → (forall a . h f a → h g a)
```

this is a transformer stack, but where we work to a specification

# Chaining Transformers

So what does a stack of size 3 look like?

What's the type?

Gahh!

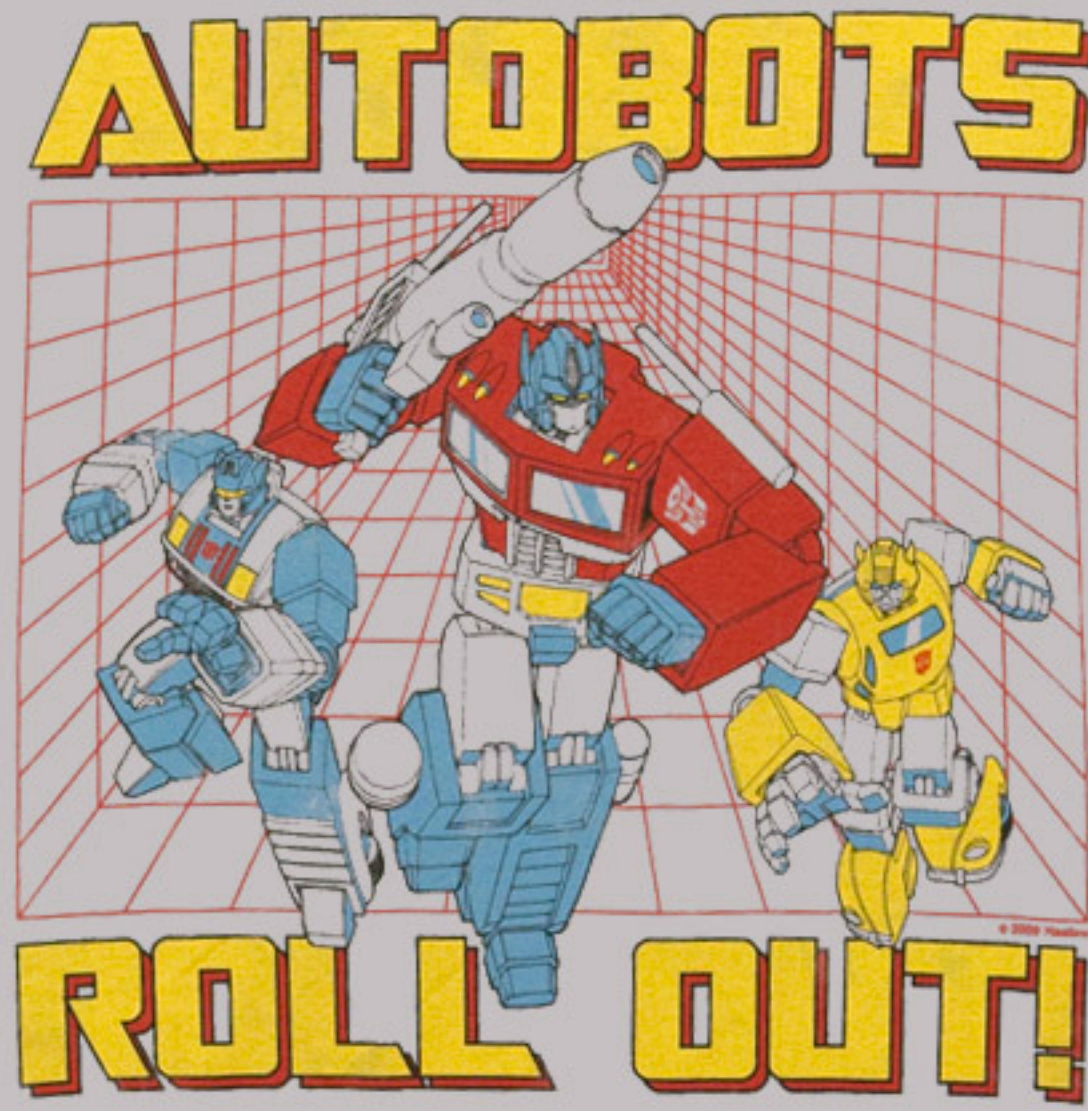
```
handleT3 ::
  (Functor g, Functor (t2 (t1 (Free g))),
   HFunctor t2, HFunctor t3,
   MonadTrans t3, MonadTrans t1, MonadTrans t2,
   MonadEff f1 (t1 (Free g)),
   MonadEff f2 (t2 (Free (CoYoneda f1 + g))),
   MonadEff f3 (t3 (Free (CoYoneda f2 + (CoYoneda f1 + g))))) ⇒

        Free (CoYoneda f3 + (CoYoneda f2 + (CoYoneda f1 + g))) a
             → t3 (t2 (t1 (Free g))) a

handleT3 = hmap (hmap handleT) . hmap handleT . handleT
```

actually, it's really not that bad: we generally have parametricity in m

```
instance Monad m ⇒ MonadEff (StateS s) (StateT s m) where
  eff (Get)   = get
  eff (Put s) = put s
```

# CONCLUSION


PROTECT
DESTROY
TRANSFORMERS


AUTOBOTS
ROLL OUT!