Witnessing Purity, Constancy and Mutability
APLAS 2009

Ben Lippmeier
Australian National University
2009/12/14

The hidden cost of state monads

```
map :: (a -> b) -> List a -> List b
map f xs
 = case xs of
     Nil       -> Nil
     Cons x xs -> Cons (f x) (map f xs)
```

The hidden cost of state monads

```
map :: (a -> b) -> List a -> List b
map f xs
 = case xs of
    Nil        -> Nil
    Cons x xs -> Cons (f x) (map f xs)
```

```
mapIO :: (a -> IO b) -> List a -> IO (List b)
mapIO f xs
 = case xs of
    Nil -> return Nil
    Cons x xs
     -> do x'  <- f x
           xs' <- mapIO f xs
           return (Cons x' xs')
```

The hidden cost of state monads

- We need two copies of every higher order function.
- It is possible to derive the non-monadic version from the monadic one, but in practice people rarely do.
- Library writers tend not to provide both versions.

Functions from Data.Map

```
adjust
  :: Ord k
  => (a -> a) -> k -> Map k a -> Map k a


mapWithKey
  :: (k -> a -> b) -> Map k a -> Map k b


mapAccumWithKey
  :: (a -> k -> b -> (a, c))
  -> a -> Map k b -> (a, Map k c)
```

- The library doesn't provide monadic versions of any of these functions...

Haskell lacks mutability polymorphism

```
data List a
 = Nil
 | Cons a (List a)
```

Haskell lacks mutability polymorphism

```
data List a
 = Nil
 | Cons a (List a)
```

```
data MList a
 = MNil
 | MCons a (IORef (MList a))
```

Haskell lacks mutability polymorphism

```
data List a
 = Nil
 | Cons a (List a)
```

```
data MList a
 = MNil
 | MCons a (IORef (MList a))

data MList2 a
 = MNil2
 | MCons2 (IORef a) (MList2 a)
```

Haskell lacks mutability polymorphism

```
data List a
  = Nil
  | Cons a (List a)
```

```
data MList a
  = MNil
  | MCons a (IORef (MList a))

data MList2 a
  = MNil2
  | MCons2 (IORef a) (MList2 a)

data MList3 a
  = MNil3
  | MCons3 (IORef a)
           (IORef (MList3 a))
```

Haskell lacks mutability polymorphism

```
data List a
 = Nil
 | Cons a (List a)
```

```
data MList a
 = MNil
 | MCons a (IORef (MList a))

data MList2 a
 = MNil2
 | MCons2 (IORef a) (MList2 a)

data MList3 a
 = MNil3
 | MCons3 (IORef a)
          (IORef (MList3 a))
```

- These four data types are all incompatible.
- How many times do you want to rewrite the definitions of **`length`**, **`map`**, and **`fold`**?

The Plan

- Use region typing, effect typing, and type classes to introduce effect and mutability polymorphism.
- Re-use the syntax and other features of Haskell.
- Types become complex, but can be inferred.
- We usually don’t write the following signatures in source programs.

The type of updateInt

```
updateInt
  :: Int -> Int -> ()
```

- **`updateInt`** uses the value of the second argument to overwrite the value of the first.

The type of updateInt

```
updateInt
  :: Int r1 -> Int r2 -> ()
```

- `updateInt` uses the value of the second argument to overwrite the value of the first.
- Region variables name what part of the store the arguments are located in.

The type of updateInt

```
updateInt
 :: forall (r1, r2 : region)
 .   Int r1 -> Int r2 -> ()
```

- **`updateInt`** uses the value of the second argument to overwrite the value of the first.
- Region variables name what part of the store the arguments are located in.

The type of updateInt

```
updateInt
 :: forall (r1, r2 : region)
 .  Int r1 -> Int r2 -(e1)> ()
 :- e1 = Read r2 \/ Write r1
```

- **`updateInt`** uses the value of the second argument to overwrite the value of the first.
- Region variables name what part of the store the arguments are located in.
- When it is called it reads its second argument and writes to the first.

Mutability constraints

```
updateInt
  :: forall (r1, r2 : region)
  .  Mutable r1 => Int r1 -> Int r2 -(e1)> ()
  :- e1 = Read r2 \/ Write r1
```

- An object in region **`r1`** is being updated, so that region must be **`Mutable`**.

Mutability constraints

```
updateInt
 :: forall (r1, r2 : region)
 .  Mutable r1 => Int r1 -> Int r2 -(e1)> ()
 :- e1 = Read r2 \/ Write r1
```

- An object in region **`r1`** is being updated, so that region must be **`Mutable`**.

- **`Mutable r1`** is a region class. (like a type class)

- To call **`updateInt`** we must pass a witness to the fact that **`r1`** really does support mutability.

Comparison with type classes

```
notEqual :: Eq a => a -> a -> Bool
```

- To call **`notEqual`**, we must pass a dictionary containing an equality operator for values of type **`a`**.
- A dictionary is evidence that a type supports a particular operation (equality in this case).
- In contrast, the evidence for the mutability of a region exists at type level, and is erased during code generation.

```
swap :: Int -> Int -> ()
```

```
swap
 = \  (x       : Int).
   \  (y       : Int).

   let tmp = 0
       _   = updateInt tmp x
       _   = updateInt x   y
       _   = updateInt y   tmp
   in  ()
```

```
swap :: Int r1 -> Int r2 -> ()


swap
 = \  (x       : Int r1) .
   \  (y       : Int r2) .

   let tmp = 0
       _   = updateInt tmp x
       _   = updateInt x   y
       _   = updateInt y   tmp
   in  ()
```

```
swap :: Int r1 -> Int r2 -(e1)> ()
     :- e1 = Read r1 \/ Read r2 \/ Write r1 \/ Write r2
```

```
swap
 = \  (x      : Int r1).
   \  (y      : Int r2).

   let tmp = 0
       _   = updateInt tmp x
       _   = updateInt x   y
       _   = updateInt y   tmp
   in  ()
```

```
swap :: forall (r1, r2 : region)
     .  Int r1 -> Int r2 -(e1)> ()
     :- e1 = Read r1 \/ Read r2 \/ Write r1 \/ Write r2
```

```
swap
 = \   (x       : Int r1).
   \   (y       : Int r2).

   let tmp = 0
       _   = updateInt tmp x
       _   = updateInt x   y
       _   = updateInt y   tmp
   in  ()
```

```
swap :: forall (r1, r2 : region)
     .  Int r1 -> Int r2 -(e1)> ()
     :- e1 = Read r1 \/ Read r2 \/ Write r1 \/ Write r2


swap
 = /\ (r1, r2 : region).
   \  (x       : Int r1).
   \  (y       : Int r2).

   let tmp = 0
       _   = updateInt tmp x
       _   = updateInt x   y
       _   = updateInt y   tmp
   in  ()
```

```
swap :: forall (r1, r2 : region)
     .  Int r1 -> Int r2 -(e1)> ()
     :- e1 = Read r1 \/ Read r2 \/ Write r1 \/ Write r2
```

```
swap
 = /\ (r1, r2 : region).
   \  (x      : Int r1).
   \  (y      : Int r2).

   let tmp = 0
       _   = updateInt tmp x
       _   = updateInt x   y
       _   = updateInt y   tmp
   in  ()
```

```
updateInt
 :: forall (r1, r2 : region)
 .  Mutable r1 => Int r1 -> Int r2 -(e1)> ()
 :- e1 = Read r2 \/ Write r1
```

```
swap :: forall (r1, r2 : region)
     .  Int r1 -> Int r2 -(e1)> ()
     :- e1 = Read r1 \/ Read r2 \/ Write r1 \/ Write r2
```

```
swap
 = /\ (r1, r2 : region).
   \  (x       : Int r1).
   \  (y       : Int r2).

   let tmp = 0
       _   = updateInt ?? ?? ?? tmp x
       _   = updateInt ?? ?? ?? x   y
       _   = updateInt ?? ?? ?? y   tmp
   in  ()
```

```
updateInt
 :: forall (r1, r2 : region)
 .  Mutable r1 => Int r1 -> Int r2 -(e1)> ()
 :- e1 = Read r2 \/ Write r1
```

```
swap :: forall (r1, r2 : region)
     .  Int r1 -> Int r2 -(e1)> ()
     :- e1 = Read r1 \/ Read r2 \/ Write r1 \/ Write r2
```

```
swap
 = /\ (r1, r2 : region).
   \  (x       : Int r1).
   \  (y       : Int r2).

   let tmp = 0
       _   = updateInt ?? r1 ?? tmp x
       _   = updateInt r1 r2 ?? x   y
       _   = updateInt r2 ?? ?? y   tmp
   in  ()
```

```
updateInt
 :: forall (r1, r2 : region)
 .  Mutable r1 => Int r1 -> Int r2 -(e1)> ()
 :- e1 = Read r2 \/ Write r1
```

```
swap :: forall (r1, r2 : region)
     .  Int r1 -> Int r2 -(e1)> ()
     :- e1 = Read r1 \/ Read r2 \/ Write r1 \/ Write r2
```

```
swap
 = /\ (r1, r2 : region).
   \  (x       : Int r1).
   \  (y       : Int r2).
   letregion r3 in
   let tmp = 0 r3
       _   = updateInt r3 r1 ?? tmp x
       _   = updateInt r1 r2 ?? x   y
       _   = updateInt r2 r3 ?? y   tmp
   in  ()
```

```
updateInt
 :: forall (r1, r2 : region)
 .  Mutable r1 => Int r1 -> Int r2 -(e1)> ()
 :- e1 = Read r2 \/ Write r1
```

```
swap :: forall (r1, r2 : region)
     .  Mutable r1 => Mutable r2
     => Int r1 -> Int r2 -(e1)> ()
     :- e1 = Read r1 \/ Read r2 \/ Write r1 \/ Write r2

swap
 = /\ (r1, r2 : region).
   /\ (w1     : Mutable r1).
   /\ (w2     : Mutable r2).
   \  (x      : Int r1).
   \  (y      : Int r2).
   letregion r3 in
   let tmp = 0 r3
       _   = updateInt r3 r1 ?? tmp x
       _   = updateInt r1 r2 w1 x   y
       _   = updateInt r2 r3 w2 y   tmp
   in  ()
```

```
updateInt
 :: forall (r1, r2 : region)
 .  Mutable r1 => Int r1 -> Int r2 -(e1)> ()
 :- e1 = Read r2 \/ Write r1
```

```
swap :: forall (r1, r2 : region)
      .   Mutable r1 => Mutable r2
      => Int r1 -> Int r2 -(e1)> ()
      :- e1 = Read r1 \/ Read r2 \/ Write r1 \/ Write r2

swap
 = /\ (r1, r2 : region).
   /\ (w1      : Mutable r1).
   /\ (w2      : Mutable r2).
   \  (x       : Int r1).
   \  (y       : Int r2).
   letregion r3 with w3 = MkMutable r3 in
   let tmp = 0 r3
       _   = updateInt r3 r1 w3 tmp x
       _   = updateInt r1 r2 w1 x   y
       _   = updateInt r2 r3 w2 y   tmp
   in  ()
```

```
MkMutable :: PI(r : region). Mutable r
```

Suspension

```
suspend
 :: forall (a, b : type) (e1 : effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b
```

- The language is call-by-value by default.
- Laziness can be introduced with the suspend operator.
- Only pure functions can be suspended.
- Any read effects must involve constant regions.

Creating witnesses of purity

```
suspend
 :: forall (a, b :: type) (e1 :: effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b

MkPurify
 :: PI(r : region). Const r -> Pure (Read r)

MkPureJoin
 :: PI(e1, e2 : effect)
 .  Pure e1 -> Pure e2 -> Pure (e1 \/ e2)
```

- Type level combinators build witnesses to the purity of an effect from witnesses to the constancy of regions.

```
mapLazy
  :: (a -> b) -> List a -> List b
```

```
mapLazy
  :: (a -> b) -> List r1 a -> List r2 b
```

```
mapLazy
  :: (a -(e1)> b) -> List r1 a -(e2)> List r2 b
  :- e2 = Read r1 \/ e1
```

```
mapLazy
  :: Pure e1
  => (a -(e1)> b) -> List r1 a -(e2)> List r2 b
  :- e2 = Read r1 \/ e1
```

```
mapLazy
  :: Pure e1 => Const r1
 => (a -(e1)> b) -> List r1 a -(e2)> List r2 b
  :- e2 = Read r1 \/ e1
```

```
mapLazy
 :: forall (a,  b  : type)
           (r1, r2 : region)
           (e1     : effect)
 .   Pure e1 => Const r1
 => (a -(e1)> b) -> List r1 a -(e2)> List r2 b
 :- e2 = Read r1 \/ e1
```

```
mapLazy
 = \  (f    : a -> b).
   \  (xx   : List a).
   case xx of
     Nil        -> Nil
     Cons x xs ->
      let x'    = f x
          mapL' = mapLazy f
          xs'   = suspend mapL' xs

      in  Cons x' xs'
```

```
mapLazy
 = \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil         -> Nil
     Cons x xs ->
      let x'     = f x
          mapL' = mapLazy f
          xs'    = suspend mapL' xs

      in  Cons x' xs'
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect)
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil        -> Nil
     Cons x xs ->
      let x'     = f x
          mapL' = mapLazy f
          xs'    = suspend mapL' xs

      in  Cons x' xs'
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect)
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil         -> Nil b r2
     Cons x xs ->
      let x'     = f x
          mapL' = mapLazy a b r1 r2 e1 f
          xs'    = suspend mapL' xs

      in  Cons b r2 x' xs'
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect)
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil          -> Nil b r2
     Cons x xs ->
      let x'     = f x
          mapL' = mapLazy a b r1 r2 e1 f
          xs'    = suspend mapL' xs

      in  Cons b r2 x' xs'
```

```
suspend
 :: forall (a, b : type) (e1 : effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect)
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil         -> Nil b r2
     Cons x xs ->
      let x'     = f x
          mapL'  = mapLazy a b r1 r2 e1 f
          xs'    = suspend ?? ?? ?? ?? mapL' xs

      in  Cons b r2 x' xs'
```

```
suspend
 :: forall (a, b : type) (e1 : effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect)
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil          -> Nil b r2
     Cons x xs -> 
      let x'     = f x
          mapL'  = mapLazy a b r1 r2 e1 f
          xs'    = suspend (List r1 a) (List r2 b)
                           ??
                           ??
                           mapL' xs

      in  Cons b r2 x' xs'
```

```
suspend
 :: forall (a, b : type) (e1 : effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect)
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil          -> Nil b r2
     Cons x xs ->
      let x'     = f x
          mapL'  = mapLazy a b r1 r2 e1 f
          xs'    = suspend (List r1 a) (List r2 b)
                            ??
                            ??
                            mapL' xs

      in  Cons b r2 x' xs'
```

Read r1

e1

```
suspend
 :: forall (a, b : type) (e1 : effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect)
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
  case xx of
     Nil         -> Nil b r2
     Cons x xs ->
      let x'     = f x
          mapL'  = mapLazy a b r1 r2 e1 f
          xs'    = suspend (List r1 a) (List r2 b)
                           (Read r1 \/ e1)
                           ??
                           mapL' xs

      in  Cons b r2 x' xs'
```

Read r1

e1

```
suspend
 :: forall (a, b : type) (e1 : effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect)
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil         -> Nil b r2
     Cons x xs ->
      let x'     = f x
          mapL'  = mapLazy a b r1 r2 e1 f
          xs'    = suspend (List r1 a) (List r2 b)
                           (Read r1 \/ e1)
                           ??
                           mapL' xs

      in  Cons b r2 x' xs'
```

```
suspend
 :: forall (a, b : type) (e1 : effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect).
   /\ (w1    : Pure  e1).
   /\ (w2    : Const r1).
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil          -> Nil b r2
     Cons x xs ->
      let x'      = f x
          mapL'   = mapLazy a b r1 r2 e1 w1 w2 f
          xs'     = suspend (List r1 a) (List r2 b)
                            (Read r1 \/ e1)
                            ??
                            mapL' xs

      in  Cons b r2 x' xs'
```

```
suspend
 :: forall (a, b : type) (e1 : effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b
```

```
mapLazy
 = /\ (a, b : type) (r1, r2 : region) (e1 : effect)
   /\ (w1    : Pure  e1).
   /\ (w2    : Const r1).
   \  (f     : a -(e1)> b).
   \  (xx    : List r1 a).
   case xx of
     Nil        -> Nil b r2
     Cons x xs ->
      let x'     = f x
          mapL'  = mapLazy a b r1 r2 e1 w1 w2 f
          xs'    = suspend (List r1 a) (List r2 b)
                           (Read r1 \/ e1)
                           (MkPureJoin
                                  (Read r1) e1
                                  (MkPurify r1 w2) w1)
                          mapL' xs
      in  Cons b r2 x' xs'
```

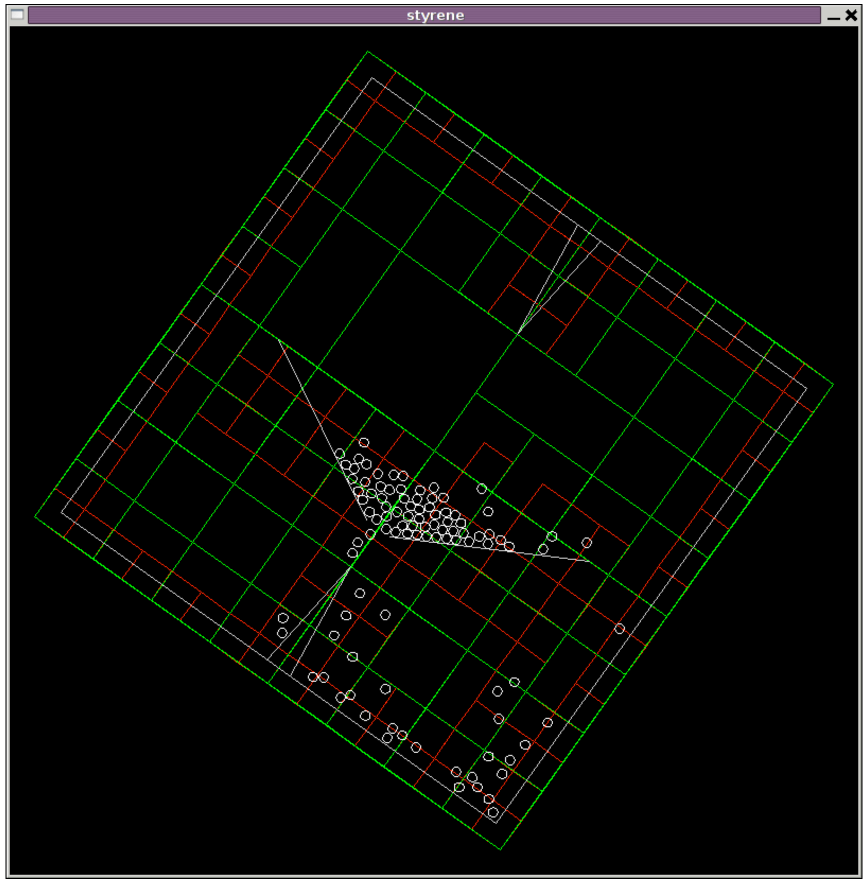
```
suspend
 :: forall (a, b : type) (e1 : effect)
 .  Pure e1 => (a -(e1)> b) -> a -> b
```

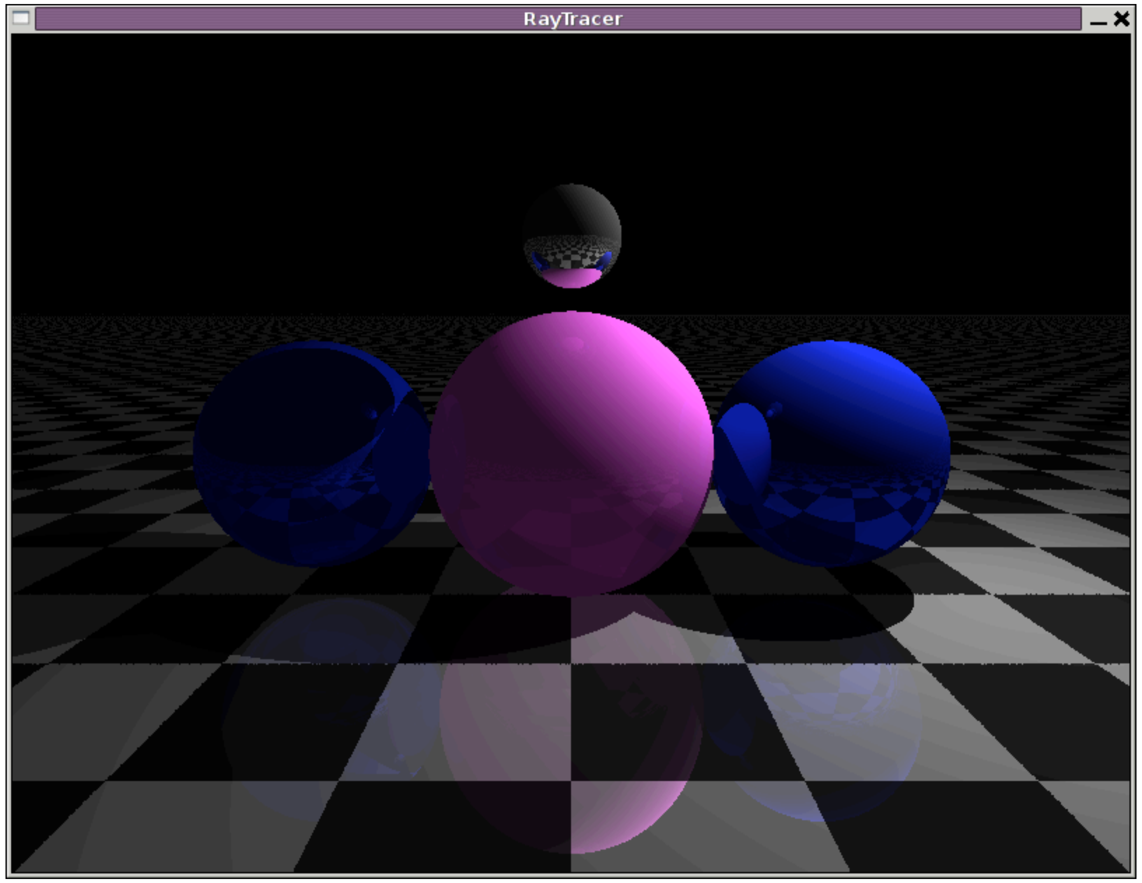
In the paper:

- Formal definition of core language
- Dynamic semantics
- Typing rules
- Soundness theorem

Implemented

- In the Disciplined Disciple Compiler (0.1.2)
- Download from **`http://trac.haskell.org/ddc`**
- “research prototype” -- help gladly accepted.





Questions?