Function Inheritance

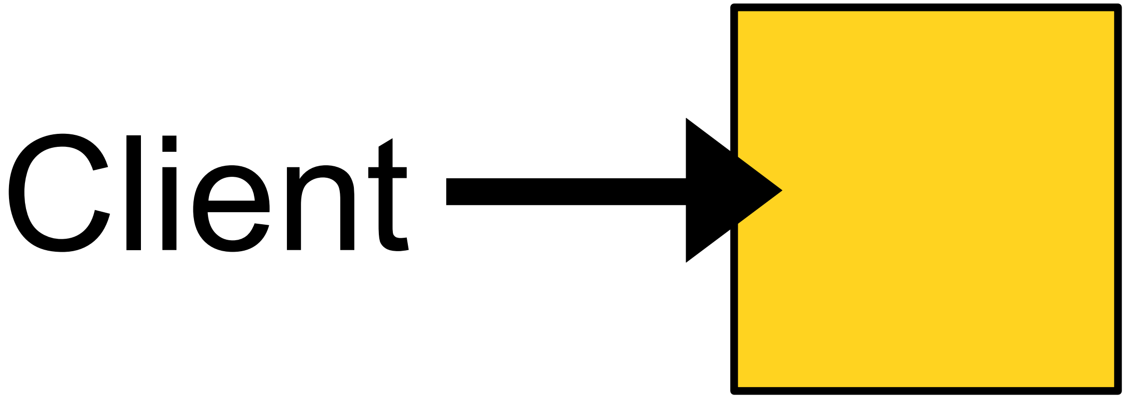
William Cook
University of Texas at Austin
Department of Computer Science
Join work with Daniel Brown, Northeastern

SBLP 2009

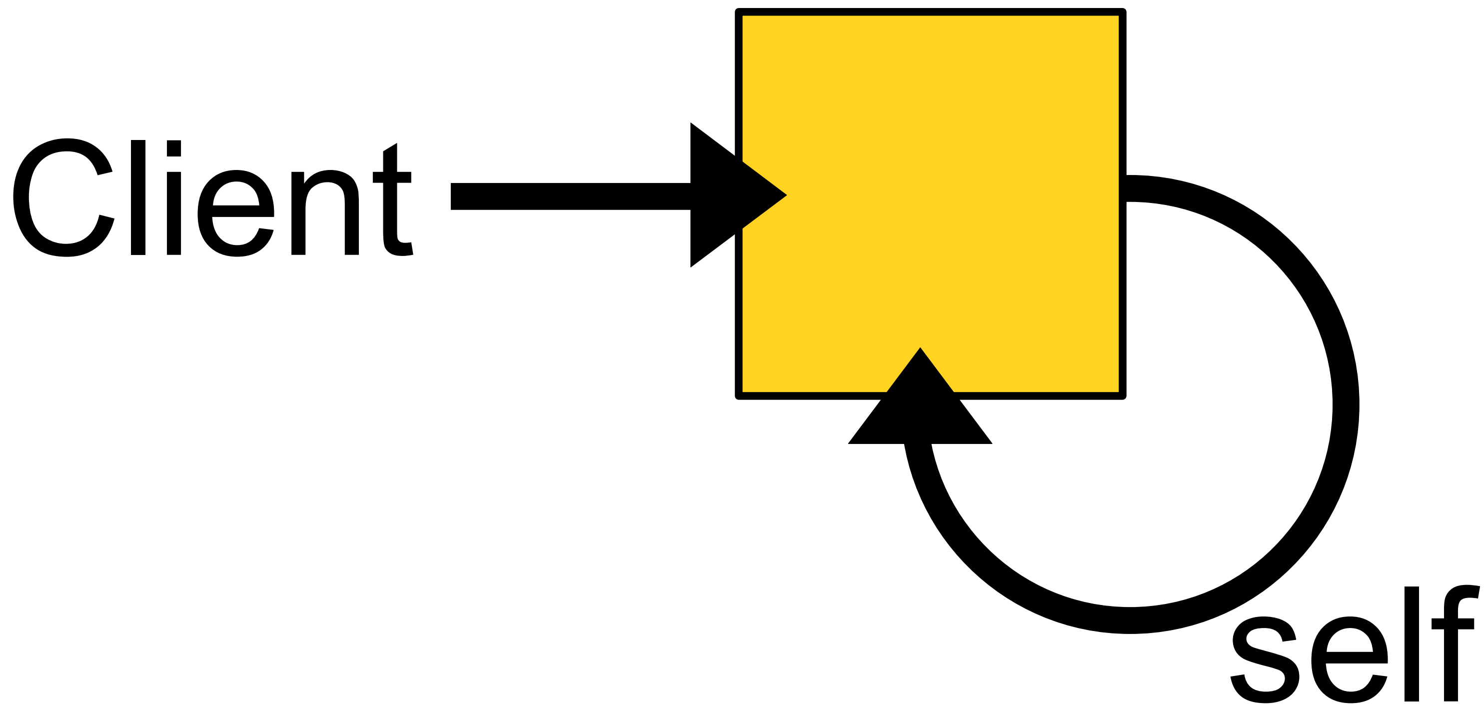
Goals

- Define "Inheritance"
  - general definition
  - captures essential characteristics
  - not specific to object-oriented programming
- Illustrate use of inheritance
  - For ***functional*** programming
  - For ***types***

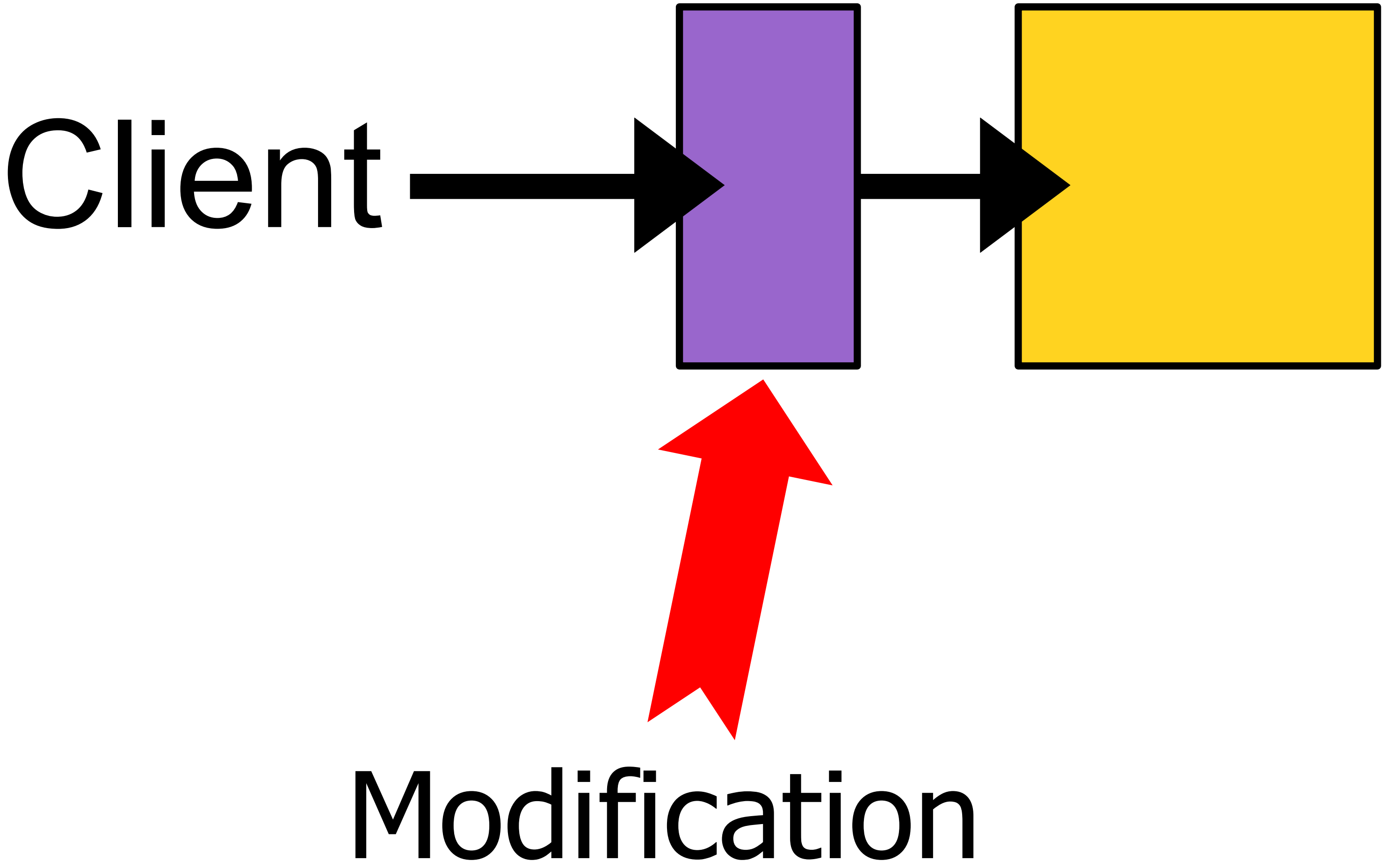
Definition



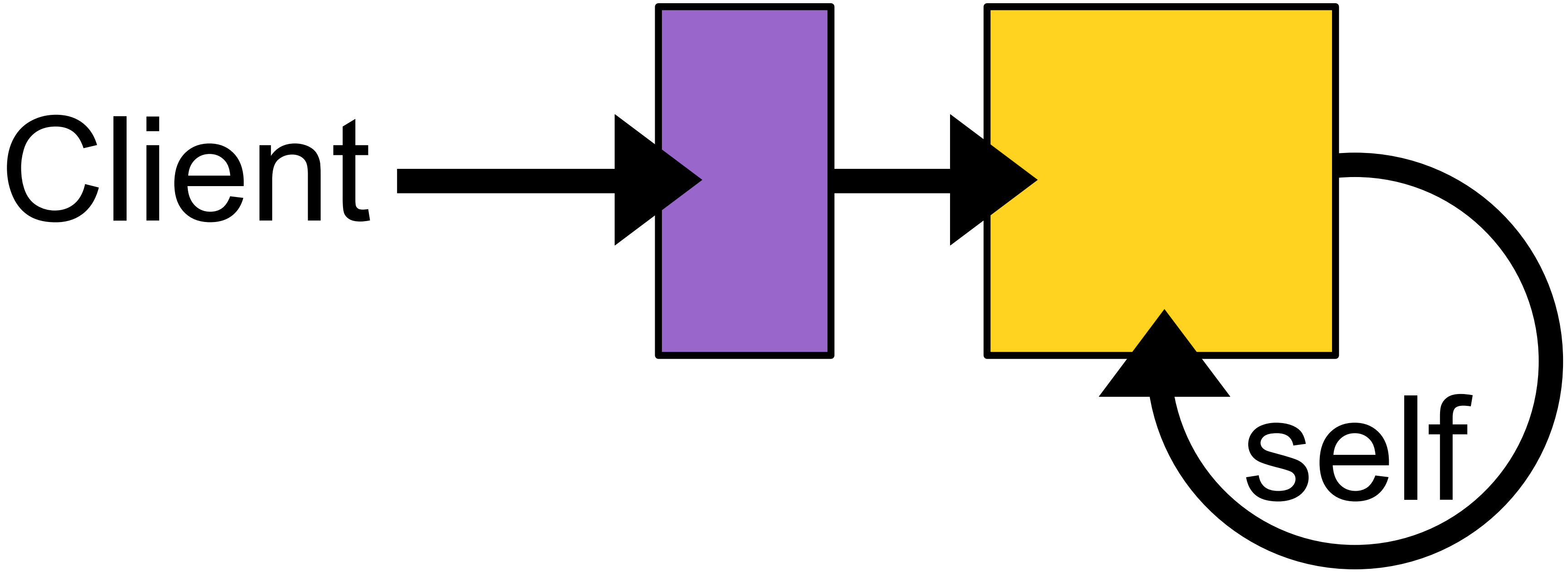
Self-reference



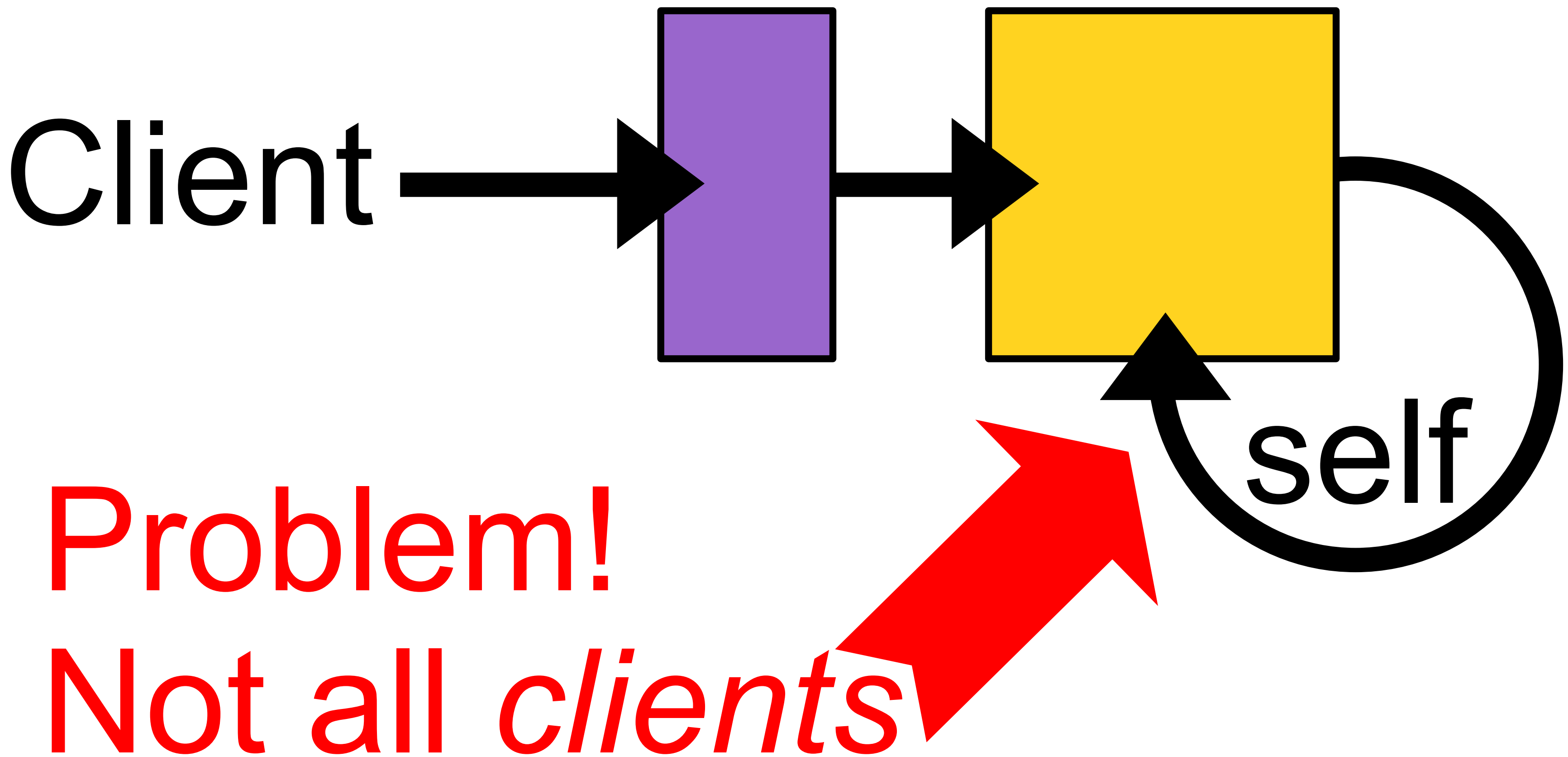
Modification



Modification and Self-reference

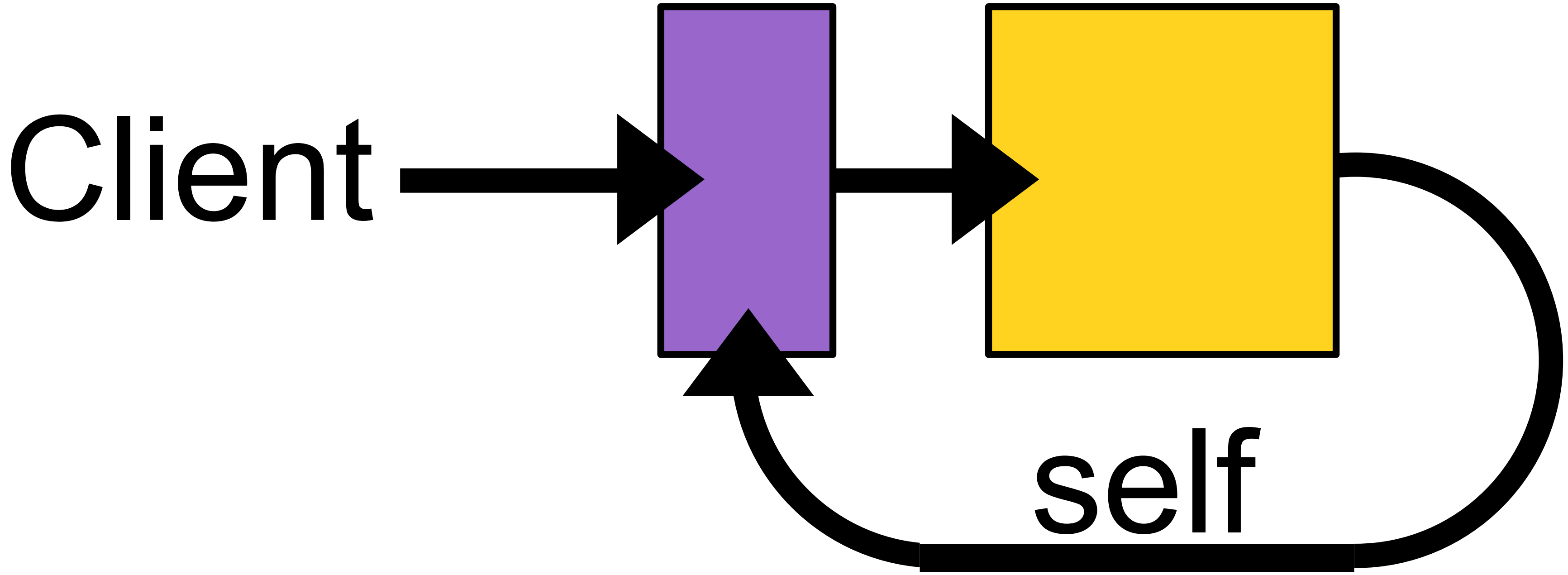


Modification and Self-reference



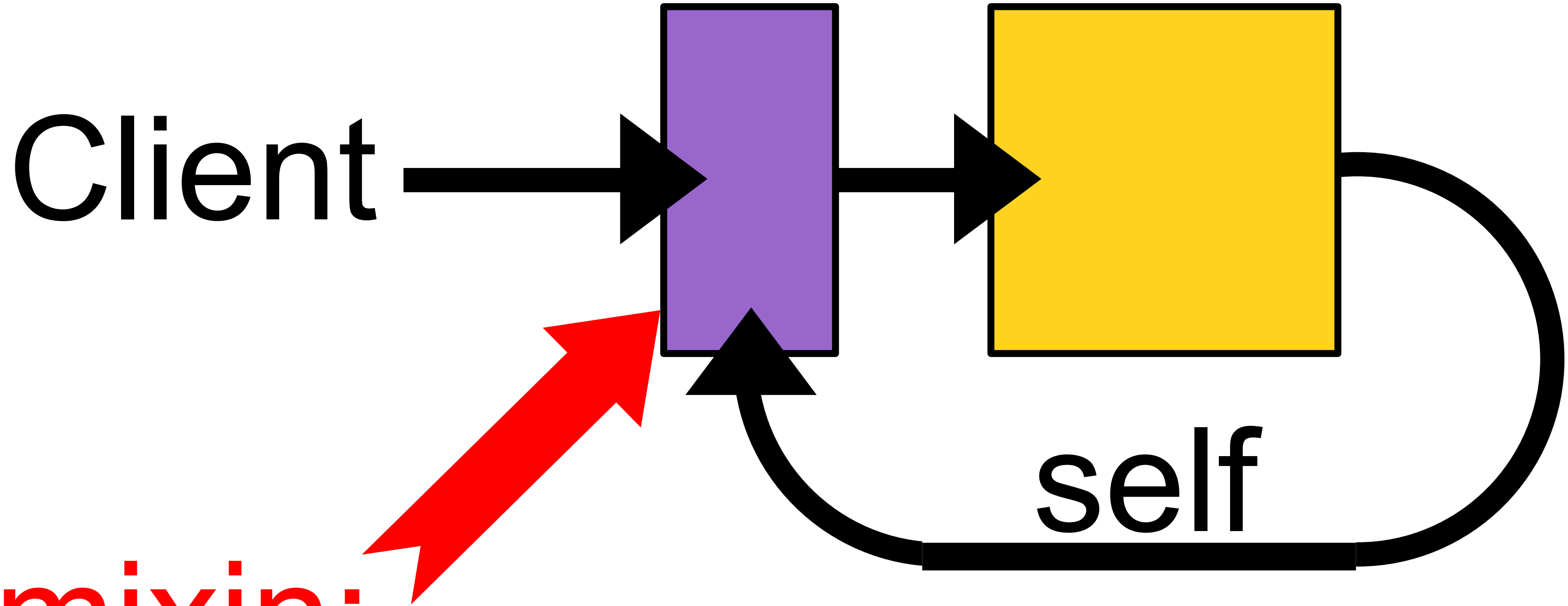
Problem!
Not all *clients*
are modified

Inheritance



Inheritance:
"Consistently modify a recursive definition"

Mixins



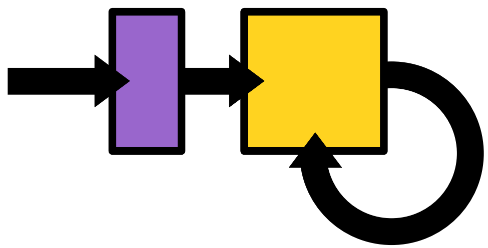
mixin:

reusable modification of self-referential structure

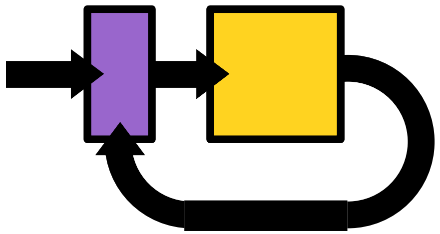
Formally

Recursion $A = Y(G)$

Modification $B = M(A)$



Inheritance $C = Y(M^\circ G)$



Where Y is standard least fixed point

In general: $M(Y(G)) \neq Y(M^\circ G)$

Inheritance

- Definitions
  - –Modify a recursive definition
  - –Composition inside fixed point Y
- Fundamentally new
  - –Many fixed points in semantics of ML, Pascal, C, textbooks
  - –Never allow composition inside Y

Observation

- Denotational semantics
  - provides strong intuition
- Operational semantics
  - examples
    - Featherweight Java [Pierce]
    - Theory of Objects [Cardelli]
  - just steps, no purpose or meaning

Standard Fibonacci

```
fib :: N → N
fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

Explicit Fixed Points

type Gen a = a → a

fix :: Gen(a) → a

fix f = f (fix f)

- creates infinite expansion

fix G = G(G(G(G(...))))

Making Self-Reference Explicit

gFib :: Gen(N → N)

gFib self 0 = 0

gFib self 1 = 1

gFib self n = self (n-1) + self (n-2)

- gFib is not recursive

fib = fix gFib

- fix gFib = gFib(gFib(gFib(...)))

A Simple Modification

mod :: Gen($N \to N$)

mod9 super 9 = 34

mod9 super n = super n

Function Inheritance

mod :: Gen(N → N)

mod9 super 9 = 34

mod9 super n = super n

fib9 = mod9 fib = mod9 (fix gFib)

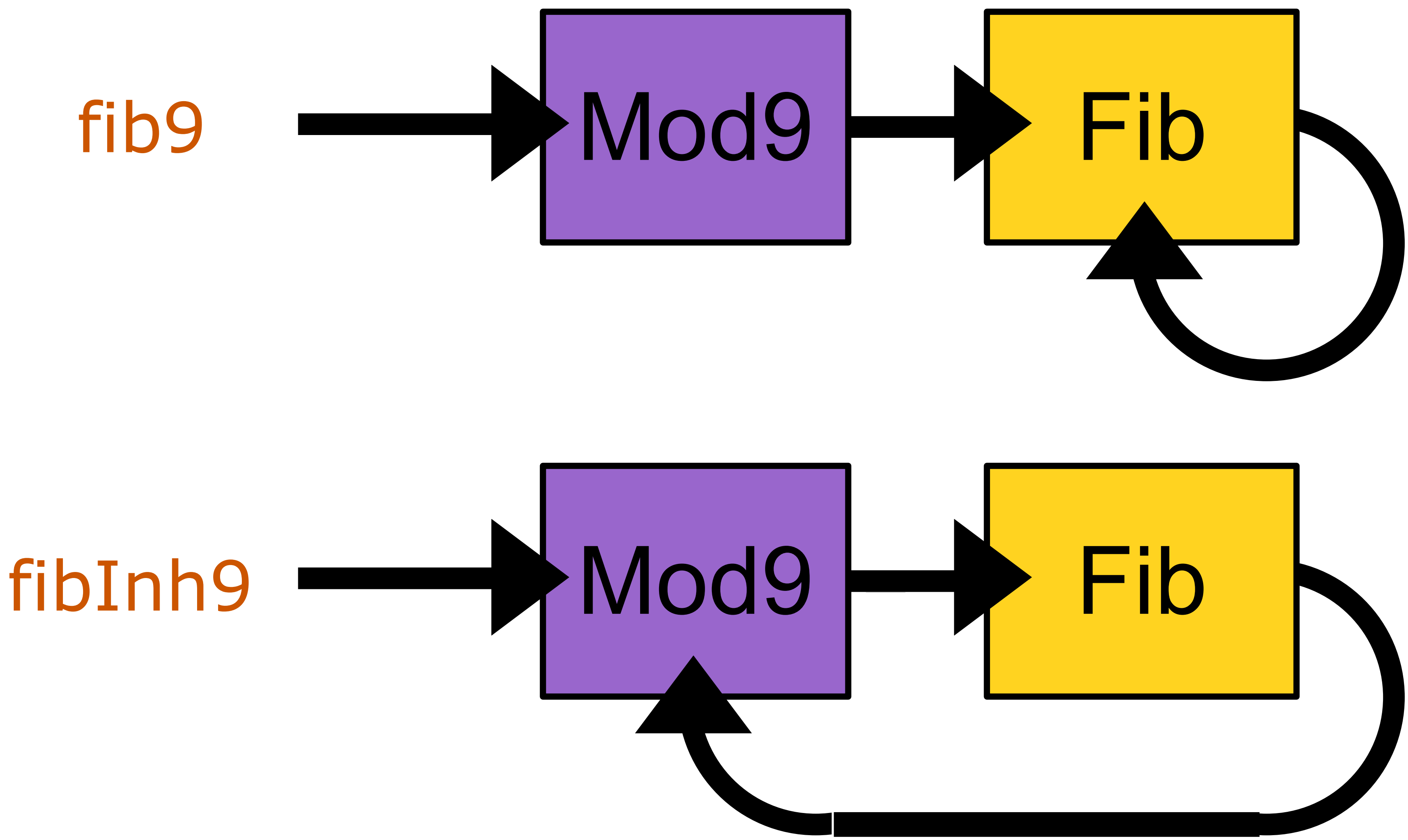
optimize computation of just fib 9

Inheritance

fibInh9 = fix(mod9 . gFib)

optimize computation for all n > 9

Summary



Monads

- Composable Computations
  - state-based computations
  - computations that can fail (exceptions)
  - etc...
- Monad hides the details

Monads

- Simple computation that produces value n:

  return n

- Compound computation:

  do

  $v_1 \leftarrow c_1$

  $v_2 \leftarrow c_2$

  …

  c_n

  – with hidden state/errors/etc
  – Looks like an imperative program

Monadification

Parameterize by arbitrary monad

```
gmFib :: Monad m => Gen(N → m N)
gmFib self 0 = return 0
gmFib self 1 = return 1
gmFib self n = do
                 a ← self (n-1)
                 b ← self (n-2)
                 return (a + b)


fibM n = runIdentity (fix gmFib n)
```

runs gmFib with an no-op monad

Memoization Mixin

```
memo :: MonadState (Map a b) m => Gen(a → m b)
memo super a = do
  b ← gets (lookup a)
  case b of
    Just b     → return b
    Nothing    → do
        b ← super a
        modify (insert a b)
        return b
```

```
class MonadState s m where
  gets      :: (s → a) → m a
  modify    :: (s → s) → m ()
```

Memoized Fibonacci

memoMapFib :: N → State (Map N N) N

memoMapFib = fix (memo . gmFib)

Inheritance

fibMap :: N → N

fibMap n = evalState (memoMapFib n) empty

Another example: Logging

```
log :: (Show a, MonadWriter String m) =>
    String → Gen(a → m b)
log name super a = do
    tell (name ++ "(" ++ show a ++ ")\n")
    super a
```

Inheritance

```
logFib = fix (log "Fib" . gmFib)
```

– Prints "Fib(3)" etc for each recursive call

Composing Mixins

mixins

```
logMemoFib = fix (memo . log "Fib" . gmFib)
```

- combine logging and memoization
- technical details:
  merge State and Writer monads

Type Inheritance

data Tree = Tree Int Tree Tree

– Add a String label at all levels of tree

data Labeled = Tree Int *Labeled Labeled* ***String***

Type Inheritance

data GTree self = GTree Int self self

data Tree = Tree (GTree Tree)

data Labeled = Lab (GTree Labeled) String

- *messy in Haskell*

Inherited Types => Inherited Funtions

```
printGTree self (GTree n t1 t2) = do
  print n
  self t1
  self t2

printTree (Tree t) = printGTree printTree t

printLabTree (Lab t lab) = do
  print lab
  printGTree printLabTree t
```

Syntax Support

- Haskell has syntax support for self-reference
  - fib n = fib(n-1) + fib(n-2)
- Syntax support for inheritance?
  - memoFib = memo **inherit** fib
  - Eliminate explicit (and messy) use of "fix"

Pointers to other work

- "Memoization Mixins" technical report
  - Full details, larger parsing example
- Feature-Oriented Programming
  - Don Batory: Mixin Layers
- Aspect-Oriented Programming
  - EffectiveAdvice: Disciplined Advice with Explicit Effects
    - joint work with Bruno Oliveira & Tom Schrijvers
- Foundations of Objects
  - On Understanding Data Abstraction, Revisited
    - Onward! Essay 2009

Summary

- Inheritance = "modify recursive structure"

- Inheritance can be used in
  - functional programming
  - logic programming
  - procedural programming
- Inheritance for
  - types, functions, procedures, modules, classes, specifications, grammars, makefiles, mutual recursion...