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

Client →
Modification

Client

Modification
Modification and Self-reference

Client

self
Problem! Not all clients are modified
Inheritance:
“Consistently modify a recursive definition”
Mixins

Client

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

\[
\text{fib} :: \mathbb{N} \rightarrow \mathbb{N} \\
\text{fib } 0 = 0 \\
\text{fib } 1 = 1 \\
\text{fib } n = \text{fib}(n-1) + \text{fib}(n-2)
\]
Explicit Fixed Points

type Gen a = a \rightarrow a
fix :: Gen(a) \rightarrow a
fix f = f (fix f)

– creates infinite expansion
fix G = G(G(G(G(\ldots))))
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

\text{mod} :: \text{Gen}(N \rightarrow N)
\text{mod}_9 \text{ super } 9 = 34
\text{mod}_9 \text{ super } n = \text{ 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

fibInh9 = fix(mod9 . gFib)
  optimize computation for all n > 9
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

```haskell
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)
```

```
fibMap :: N → N
fibMap n = evalState (memoMapFib n) empty
```
Another example: Logging

\[
\text{log} :: \left(\text{Show } a, \text{MonadWriter String } m\right) \Rightarrow \\
\text{String} \rightarrow \text{Gen}(a \rightarrow m \, b) \\
\text{log } \text{name } \text{super } a = \text{do} \\
\text{tell } (\text{name } ++ "\((" ++ \text{show } a ++ "\))\n") \\
\text{super } a
\]

\[\text{logFib} = \text{fix } (\text{log } "\text{Fib}" \cdot \text{gmFib})\]

\[\text{– Prints } \text{“Fib(3)” etc for each recursive call}\]
Composing 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 Functions

\[\text{printGTree self (GTree n t1 t2)} = \text{do}\]
\[\quad \text{print n}\]
\[\quad \text{self t1}\]
\[\quad \text{self t2}\]

\[\text{printTree (Tree t)} = \text{printGTree printTree t}\]

\[\text{printLabTree (Lab t lab)} = \text{do}\]
\[\quad \text{print lab}\]
\[\quad \text{printGTree printLabTree t}\]
Syntax Support

• Haskell has syntax support for self-reference
  – fib n = fib(n-1) + fib(n-2)

• Syntax support for inheritance?
  – memoFib = memo \texttt{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...