EffectiveAdvice

Disciplined Advice with Explicit Effects

Bruno C. d. S. Oliveira (bruno@ropas.snu.ac.kr)
ROSAEC Center, Seoul National University

(joint work with Tom Schrijvers & William R. Cook)

Motivation

- Several modularity approaches
  - Object Oriented Programming Inheritance (OOP)
  - Aspect Oriented Programming (AOP)
  - Feature Oriented Programming (FOP)

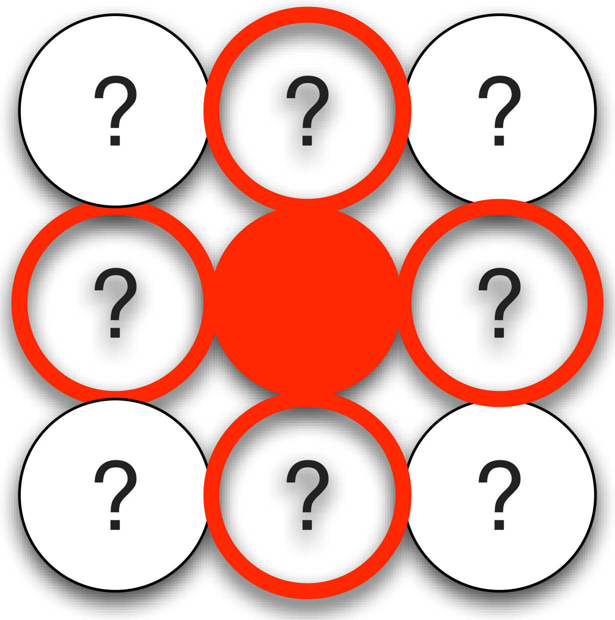
all use inheritance in some way

- are hard to understand, due to
  - hidden control flows
  - hidden data dependencies
- Main Challenges: modular reasoning and reasoning about interference between components.

Reasoning Properties

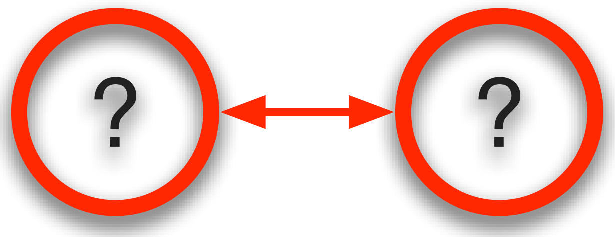
1. <u>Modular Reasoning</u>

   understand component individually



2. <u>Interference</u>

   understand interaction between components



Previous Work

- Kiczales & Mezini:

  "modular reasoning for AOP and similar mechanisms is hard"

- Aldrich:

  modular reasoning is possible for an *effect-free* approach (Open Modules)

Goals

- reason modularly about tightly coupled components
- using familiar reasoning techniques like equational reasoning and parametricity.
- understand the essence of AOP-like advice.

EffectiveAdvice

- model of AOP advice using open recursion
  - no new calculus, just System F (Haskell for surface syntax)
  - based on Cook's (1989) denotational semantics of inheritance
- full support for effects via monads
- reasoning about interference
  - harmless advice/inheritance theorems

Advice and Open Recursion

Open Recursion

```
type Open s = s -> s
```

fixpoint

```
weave :: Open s -> s
weave a = a (weave a)
```

function composition

```
(⊕) :: Open s -> Open s -> Open s
a1 ⊕ a2 = \proceed -> a1 (a2 proceed)
```

identity function

```
zero :: Open s
zero = \proceed -> proceed
```

Example

```
fib₁ :: Open (Int -> Int)
fib₁ proc n = case n of
 0 -> 0
 1 -> 1
 _ -> proc (n-1) + proc (n-2)

advfib :: Open (Int -> Int)
advfib proc n = case n of
 10 -> 55
 _   -> proc n

slowfib, optfib :: Int -> Int
slowfib = weave fib₁
optfib  = weave (advfib ⊕ fib₁)
```

Open Recursion

$$\begin{tikzcd}[column sep=small]
p = weave\ (& a_1 \arrow[rr, bend right, "proceed"'] & \oplus & a_2 & \oplus \ldots \oplus & a_n \arrow[rr, bend right, "proceed"'] & \oplus & base\,) \arrow[llllll, bend right, "proceed"']
\end{tikzcd}$$

Note: proceed ~ super in OOP

Effects

Effects

- Cook’s model of inheritance is purely functional (no side-effects)
- this is great for reasoning, but ...
- any realistic examples require effects
- Solution: use monads!

Example: Modular Memoization

may have side-effects

```
fib2 :: Monad m => Open (Int -> m Int)
fib2 proc n = case n of
 0 -> return 0
 1 -> return 1
 _ -> do x <- proc (n-1)
         y <- proc (n-2)
         return (x + y)

slowfib2 :: Int -> Int
slowfib2 = runId . weave fib2
```

recovering naive fib

Example: Modular Memoization

side-effect

```
memo :: MonadState Map m => Open (Int -> m Int)
memo proc n =
 do map <- get
    if member n map
     then return (map ! n)
     else do r <- proc n
             map' <- get put (insert n r map')
             return r

optfib2 :: Int -> Int
optfib2 n = evalState (weave (memo ⊕ fib2) n)
                          emptyMap
```

Interference

Disciplining Advice for Reasoning

- control and data flow combinators inspired by Rinard et al. (2004).
- exploit purity for reasoning
  - equational reasoning
  - parametricity
- modular non-interference proofs.

Control Flow Interference

- Cassification of Rinard et al.

  - **Combination**: An advice can call *proceed* any number of times.
  - **Replacement**: There are no calls to *proceed* in advice.
  - **Augmentation**: An advice that calls *proceed* exactly once, and does not modify the arguments to *proceed* or the value returned by *proceed*.
  - **Narrowing**: An advice that calls *proceed* at most once, and does not modify the arguments to *proceed* or the value returned by *proceed*.

Combinators: Example

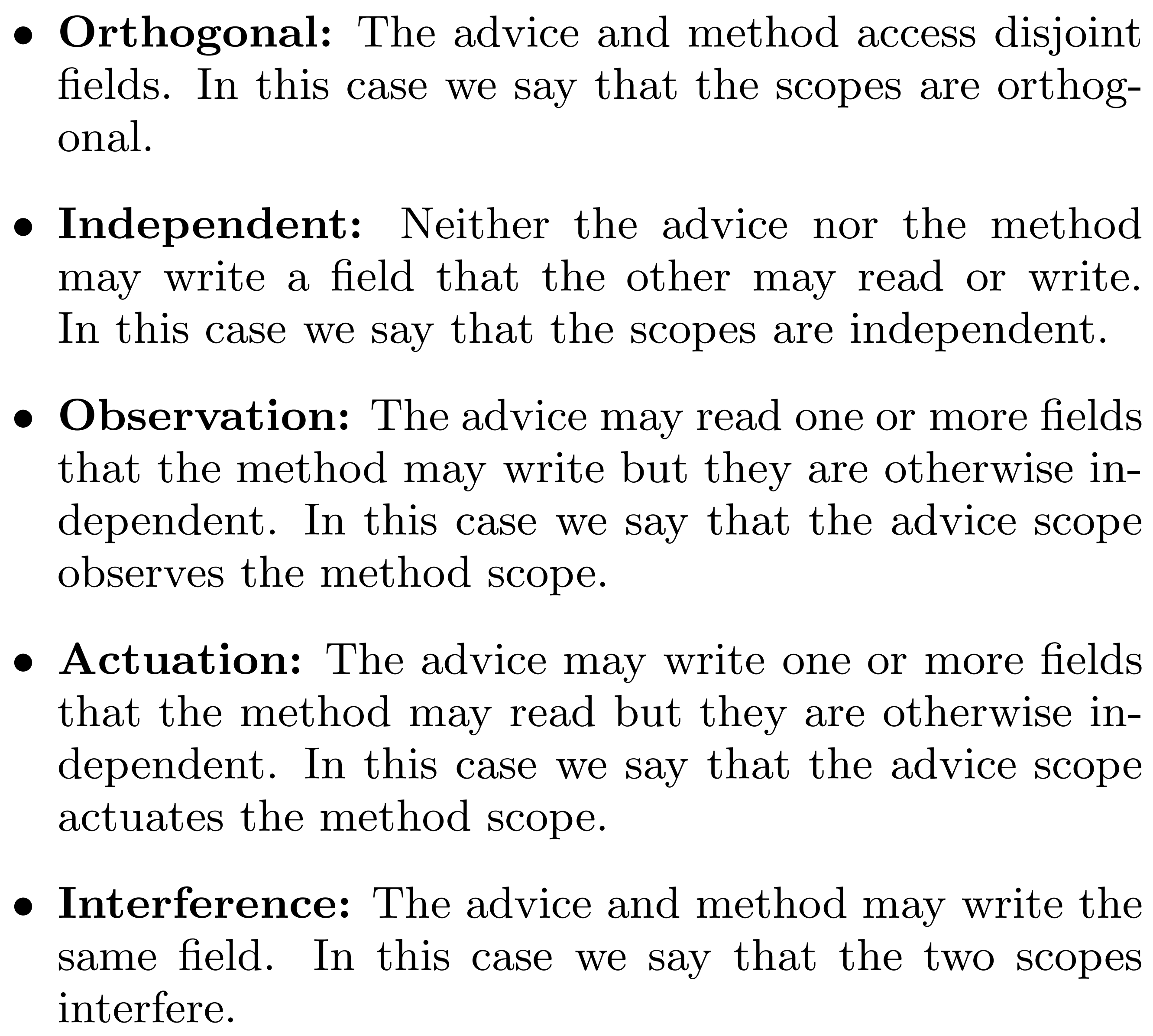
```
type Augment a b c m =
  (a -> m c, a -> b -> c -> m ())

augment :: Monad m => Augment a b c m
-> Open (a -> m b)
augment (before, after) proc a =
 do c <- before a
    b <- proc a
    after a b c
    return b
```

In words: proceed should be called once and only one.

Dataflow/Effect Interference

- Classification of Rinard et al.



- **Orthogonal:** The advice and method access disjoint fields. In this case we say that the scopes are orthogonal.
- **Independent:** Neither the advice nor the method may write a field that the other may read or write. In this case we say that the scopes are independent.
- **Observation:** The advice may read one or more fields that the method may write but they are otherwise independent. In this case we say that the advice scope observes the method scope.
- **Actuation:** The advice may write one or more fields that the method may read but they are otherwise independent. In this case we say that the advice scope actuates the method scope.
- **Interference:** The advice and method may write the same field. In this case we say that the two scopes interfere.

Decomposition of Non-Interference

```
-- * Advice
-- ** Interfering:
type IAdvice a b m
  = Open (a -> m b)
-- ** Non-Interfering:
type NIAdvice a b t
  = forall m. (Monad m, Monad (t m))
    => Open (a -> t m b)
-- * Base Component
-- similar classification
```

Interference Combinators

- 4 possible combinations

$$adv \ominus bse = niadvice\ adv \oplus nibase\ bse$$

$$adv \ogreaterthan bse = adv \oplus nibase\ bse$$

$$adv \olessthan bse = niadvice\ adv \oplus bse$$

$$adv \otimes bse = adv \oplus bse$$

Harmless Advice

Harmless Advice

- Dantas & Walker:

A piece of harmless advice is a computation that, like ordinary aspect-oriented advice, executes when control reaches a designated control-flow point. However, unlike ordinary advice, harmless advice is designed to obey a weak non-interference property. Harmless advice may change the termination behavior of computations and use I/O, but it does not otherwise influence the final result of the mainline code.

Harmless Advice

- Non-interference along 2 axes
  - control flow: augmentation
  - data flow: orthogonal

$$adv \circledast bse = augment\ adv \ominus bse$$

Harmless Advice Theorem

$$proj \circ (weave\ (adv \circledast bse)) \equiv runIdT \circ (weave\ bse)$$

given any projection function s. t.

$$proj :: \forall m, a. Monad\ \ m\ \Rightarrow\ \tau\ \ m\ \ a \rightarrow\ \ m\ \ a$$

$$proj \circ \mathit{lift} \equiv id$$

Harmless Effects

```
-- * Writer
projW :: forall w m a. (Monoid w, Monad m)
      => WriterT w m a -> m a
projW m = do (r,w) <- runWriterT m
             return r
-- * State
projS :: forall s m a. Monad m
      => s -> StateT s m a -> m a
projS s0 m = do (r,sn) <- runStateT m s0
                return r
```

Harmfull Effects

```
-- * Exceptions

projE :: forall e m a. Monad m
      => ErrorT e m a -> m a
projE m = runErrorT m >>= \x -> case x of
  Right r     -> return r
  Left error -> ???

-- * IO
```

Harmless Observation Advice

- Relaxation of harmlessness notion
- Observation advice does not affect the base computation

$$adv \odot bse = augment\ adv\ `observation`\ bse$$

$$proj \circ (weave\ (adv \odot bse)) \equiv runIdT \circ (weave\ bse)$$

Proof Technique

- We need to know only the types, not the implementations of
  - advice,
  - base component, and
  - projection function

Proof Technique

- We do rely on
  - definitions of the combinators
  - monad laws
  - purity of Haskell

Parametricity

- Parametric polymorphism ...
- ... yields Theorems for Free!

```
f :: forall a. [a] -> Int

l   :: [A]
g   :: A -> B
map :: forall a b. (a -> b) -> [a] -> [b]

          f l == f (map g l)
```

Related Work

- Various works on reasoning for AOP
  - global analysis [Kiczales&Menzini]
  - type-and-effect system on impure ML calculus (weaker) [Dantas&Walker]
  - ...
- EffectiveAdvice: very light-weight, yet strong results, combinators

Conclusion

- EffectiveAdvice: disciplined advice with explicit effects, allows
  - modular reasoning
  - reasoning about interference

Questions?

Augmentation Example

```
log :: ... => String -> Augment a b () m
log name = (bef,aft) where
  bef a = write “Entering ” a
  aft b = write “Exiting ” b
  write msg x = ...

eval = weave (augment log “evaluator” ⊕
basic_eval)
```