EffectiveAdvice: Overview, background and proofs

Bruno C. d. S. Oliveira
Tom Schrijvers
William R. Cook

Report CW 556, July 2009



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

EffectiveAdvice: Overview, background and proofs

Bruno C. d. S. Oliveira
Tom Schrijvers
William R. Cook

Report CW 556, July 2009

Department of Computer Science, K.U.Leuven

Abstract

Advice is a mechanism, widely used in aspect-oriented languages, that allows one program component to augment or modify the behavior of other components. Advice is useful for modularizing concerns, including logging, error handling, and some optimizations, that would otherwise require code to be scattered throughout a system. When advice and other components are composed together they become tightly coupled, sharing both control and data flows. However this creates important problems: *modular reasoning* about a component becomes very difficult; and two tightly coupled components may *interfere* with the control and data flows of each other.

This paper presents *EffectiveAdvice*, a disciplined model of advice, inspired by Aldrich's *Open Modules*, that has full support for effects in both base components and advice. With EffectiveAdvice, equivalence of advice, as well as base components, can be checked by *equational reasoning*. The paper describes an implementation of EffectiveAdvice as a Haskell library and shows how to use it to solve well-known programming problems. Advice is modeled by mixin inheritance and effects are modeled by monads. Interference patterns previously identified in the literature are expressed as combinators. *Parametricity*, together with the combinators, is used to prove two *harmless advice* theorems. The result is an effective model of advice that supports effects in both advice and base components, and allows these effects to be separated with strong non-interference guarantees, or merged as needed.

Keywords : AOP, effects, AOP, monads, Haskell, mixins, interference, harmless advice

CR Subject Classification : D.2.3 Coding Tools and Techniques, D.3.3 Language Constructs and Features

EffectiveAdvice: Overview, background and proofs

Bruno C. d. S. Oliveira [*]
University of Oxford, UK
bruno@comlab.ox.ac.uk

Tom Schrijvers [†]
Katholieke Universiteit Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

William R. Cook [‡]
University of Texas at Austin, USA
wcook@cs.utexas.edu

Abstract

Advice is a mechanism, widely used in aspect-oriented languages, that allows one program component to augment or modify the behavior of other components. Advice is useful for modularizing concerns, including logging, error handling, and some optimizations, that would otherwise require code to be scattered throughout a system. When advice and other components are composed together they become tightly coupled, sharing both control and data flows. However this creates important problems: *modular reasoning* about a component becomes very difficult; and two tightly coupled components may *interfere* with the control and data flows of each other.

This paper presents *EffectiveAdvice*, a disciplined model of advice, inspired by Aldrich's *Open Modules*, that has full support for effects in both base components and advice. With EffectiveAdvice, equivalence of advice, as well as base components, can be checked by *equational reasoning*. The paper describes an implementation of EffectiveAdvice as a Haskell library and shows how to use it to solve well-known programming problems. Advice is modeled by mixin inheritance and effects are modeled by monads. Interference patterns previously identified in the literature are expressed as combinators. *Parametricity*, together with the combinators, is used to prove two *harmless advice* theorems. The result is an effective model of advice that supports effects in both advice and base components, and allows these effects to be separated with strong non-interference guarantees, or merged as needed.

1. Introduction

In many specialized forms of modularity, including *aspect-oriented programming* (AOP) (Kiczales et al. 1997), *feature-oriented programming* (FOP) (Prehofer 1997), and *object-oriented programming* (OOP) inheritance (Cook 1989), the control flow and data dependencies between components are quite complex. In all these systems, *open recursion* allows control to flow back and forth between modular components during composition, making these components semantically tightly coupled despite being textually separated. This makes reasoning a significant challenge: it is hard to understand a component in isolation, and it is hard to understand the interaction between components. The former problem is known as *modular reasoning* and it has been intensely studied in both the OOP and AOP literature (Stata and Guttag 1995; Kiczales and Mezini 2005; Aldrich 2005). The latter problem, usually referred to as *interference*, has also received much attention in the AOP literature (Rinard et al. 2004; Douence et al. 2004; Dantas and Walker 2006; Clifton et al. 2007). The essence of both problems lies in the hidden *control* and *data* flows, required by the tight coupling of components, but not visible from the interfaces of these same components.

Advice is a mechanism for one program component to augment or modify the behavior of other components, which is widely used in AOP. It is useful to capture so-called *crosscutting* concerns such as logging, error handling or some optimizations. Advice provides a good example of the problems of combining open recursion and effects, since the mechanism creates tight couplings between advice and the advised programs. Note that, for the purposes of this paper, the advice mechanism *does not* correspond to the *pointcut-advice* model (Wand et al. 2004), in which advice declarations are defined together with *pointcut* declarations, allowing AOP languages to automatically apply the advice to several points in the program. Instead, we mean a model of explicit advice composition, exposing the basic mechanism of open recursion, rather than relying on the aspect language to compose advice automatically. There is no loss of expressiveness with respect to the pointcut-advice model, only loss of convenience as advice must be added individually to all points of interest. In any case, pointcuts are largely orthogonal to the problem of reasoning about open recursion, which is the main topic of this paper.

Kiczales and Mezini (2005) argue that modular reasoning about AOP, and similar mechanisms that capture crosscutting concerns, is not possible, and that a degree of global analysis is always needed. In contrast, Aldrich (2005) presents a pure functional kernel language for advice that does support modular reasoning. A key contribution of Aldrich's work is the idea that a component should control the points where it can be advised and declare these points in a public contract. Aldrich's solution for reasoning about tightly coupled components is simple: he proposes a purely functional core language, which does not allow any effects, removing the biggest obstacle to reasoning. Unfortunately, this solution is not effective in practice, as almost all practical uses of advice involve effects, and many programs subject to advice also use effects. Aldrich tries to address this practical limitation by adding an ad-hoc notion of state, however, it is no longer clear how stateful advice should be reasoned about, since state is not considered by his formal framework.

This paper presents *EffectiveAdvice*, a disciplined model of advice that is inspired by Aldrich's Open Modules and retains similar reasoning properties. However, unlike Open Modules, effects are fully supported in both base programs and advice. EffectiveAdvice promotes the idea that effects should be an integral part of the interfaces of components, and that no implicit effects should occur.

[*] Post-doctoral researcher supported by the EPSRC grant EP/E02128X.

[†] Post-doctoral researcher of the Fund for Scientific Research - Flanders.

[‡] Supported by NSF grants CCF-0448128, CCF-0724979 and CCF-6752487.

[Copyright notice will appear here once 'preprint' option is removed.]

$$\textbf{type}\ \mathit{Open}\ s = s \to s$$
$$\mathit{weave} :: \mathit{Open}\ s \to s$$
$$\mathit{weave}\ a = a\ (\mathit{weave}\ a)$$
$$\mathit{zero} :: \mathit{Open}\ s$$
$$\mathit{zero} = \mathit{id}$$
$$(\oplus) :: \mathit{Open}\ s \to \mathit{Open}\ s \to \mathit{Open}\ s$$
$$a_1 \oplus a_2 = \lambda \mathit{proceed} \to a_1\ (a_2\ \mathit{proceed})$$

Figure 1. Basic mixin combinators.

The programming model is based on open recursion, explicit advice points, and a requirement for every component to state the effects that it uses. An implementation of EffectiveAdvice as a Haskell library is elaborated; *mixin composition* is used to weave advice into a base program (Cook 1989). Monads (Wadler 1992) model effects and, for compositionality, non-monadic functions must be lifted into a monad. A second implementation in Scala is also briefly illustrated.

In the purely functional model for EffectiveAdvice, equivalence of advice, as well as base programs, is determined by *equational reasoning*. Different interference patterns (Rinard et al. 2004) between advice and base programs, constraining possible data and control flow interactions, can be enforced through the use of combinators. *Higher-rank types* (Peyton Jones et al. 2007) are used to ensure non-interference of effects. A key novelty introduced by EffectiveAdvice is the use of *parametricity* (Wadler 1989; Voigtländer 2009) to prove theorems for combinators providing strong guarantees of non-interference. In particular, two *harmless advice* (Dantas and Walker 2006) theorems are proved in this work.

In summary, the contributions of this paper are:

- EffectiveAdvice: A disciplined model of advice with full support for effects in both base programs and advice. In EffectiveAdvice effects are an integral part of the interfaces of components. In the idealized programming model, familiar reasoning techniques such as *equational reasoning* and *parametricity* can be used, yet interesting programs can be expressed.
- Strong guarantees of control and data flow non-interference through the use of combinators and the type system.
- A novel use of *parametricity* to enforce and reason about non-interference of effects between components, which is used to prove theorems for *harmless advice* and *harmless observation advice*.
- An implementation of the EffectiveAdvice methodology as a Haskell library using open recursion to model advice and monads to model effects. The approach is *statically typed* and *purely functional*.

The proofs of the theorems and background information are available in appendix.

2. EffectiveAdvice

This section introduces the Haskell implementation of EffectiveAdvice using open recursion and monads.

2.1 Open Recursion

Open recursion is a property of a component in which recursive references are left open, so that the recursive behavior can be extended later. Open recursion is the basis for inheritance and mixin composition in object-oriented languages (Cook 1989). The connection between mixins and aspects is well-known (Lopez-Herrejon et al. 2006). Open recursion is easily implemented in Haskell or Scala by introducing an explicit parameter for self-reference, rather than relying on the built-in recursive naming in the language. An explicit fixpoint operation is required to convert an open recursive component into an ordinary, closed component that can be invoked.

The basis of the implementation is shown in Figure 1. The type $\mathit{Open}\ s$ is a synonym for a function with type $s \to s$ representing open recursion. The parameter of that function is called a *join point*, that is, the point in the component in which advice is added. The operation $\oplus$ defines component (or advice) composition. Composition is associative, and it has the zero component as left and right units of $\oplus$, forming a monoid. Note that this is just the monoid of endofunctions with identity and function composition.

$$f \oplus \mathit{zero} \equiv f \equiv \mathit{zero} \oplus f$$
$$(f \oplus g) \oplus h \equiv f \oplus (g \oplus h)$$

The function weave is a fixpoint combinator used for closing, or sealing, an open and potentially advised component.

Consider the following open functions:

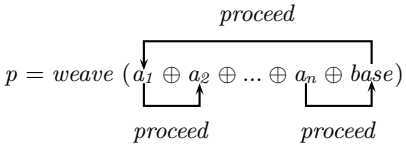
$$\begin{aligned}
&\mathit{fib}_1 :: \mathit{Open}\ (\mathit{Int} \to \mathit{Int})\\
&\mathit{fib}_1\ \mathit{proceed}\ n = \textbf{case}\ n\ \textbf{of}\\
&\quad 0 \to 0\\
&\quad 1 \to 1\\
&\quad _ \to \mathit{proceed}\ (n-1) + \mathit{proceed}\ (n-2)\\
&\mathit{advfib} :: \mathit{Open}\ (\mathit{Int} \to \mathit{Int})\\
&\mathit{advfib}\ \mathit{proceed}\ n = \textbf{case}\ n\ \textbf{of}\\
&\quad 10 \to 55\\
&\quad 30 \to 832040\\
&\quad _ \to \mathit{proceed}\ n
\end{aligned}$$

The open function fib_1 defines the standard fibonacci function, except that recursive calls are replaced by $\mathit{proceed}$. The open function advfib optimizes two calls of the fibonacci function by returning the appropriate values immediately. Note that advfib is not meant to be used standalone. It assumes that it is used in combination with an open function like fib_1 that takes care of the uncovered cases.

Different combinations of open functions are closed through weaving:

$$\begin{aligned}
&\mathit{slowfib}_1, \mathit{optfib} :: \mathit{Int} \to \mathit{Int}\\
&\mathit{slowfib}_1 = \mathit{weave}\ \mathit{fib}_1\\
&\mathit{optfib} \quad = \mathit{weave}\ (\mathit{advfib} \oplus \mathit{fib}_1)
\end{aligned}$$

The functions $\mathit{slowfib}_1$ and optfib illustrate that EffectiveAdvice unifies the concept of advice and base programs under a single type. There is still a conceptual difference between them, because in a base program $\mathit{proceed}$ is understood as a recursive call, while in advice $\mathit{proceed}$ refers to the original computation being wrapped. Weaving advice alone will typically result in a useless program, as it has no base case. This distinction becomes clearer when we visualize what happens with $\mathit{proceed}$ calls in a chain of advice being composed.



In the advice a_1 the $\mathit{proceed}$ reference is pointing to a_2 (the next advice in the chain); in the a_2 advice $\mathit{proceed}$ points to the next advice in the chain and so on for the other advice. When the base program is reached, $\mathit{proceed}$ just points back to the beginning of

```
memo :: MonadState (Map Int Int) m ⇒ Open (Int → m Int)
memo proceed x =
  do m ← get
     if member x m then return (m ! x)
                   else do y ← proceed x
                           m' ← get
                           put (insert x y m')
                           return y

fib₂ :: Monad m ⇒ Open (Int → m Int)
fib₂ proceed n = case n of
  0 → return 0
  1 → return 1
  _ → do y ← proceed (n − 1)
         x ← proceed (n − 2)
         return (x + y)
```

Figure 2. Memoization

```
runId       :: Id a → a

runState    :: State s a → s → (a, s)
runStateT   :: StateT s m a → s → m (a, s)

runWriter   :: Writer w a → (a, w)
runWriterT  :: WriterT w m a → m (a, w)

runErrorT   :: ErrorT e m a → m (Either e a)

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

class Monad m ⇒ MonadError e m | m → e where
  throwError :: e → m a
  catchError :: m a → (e → m a) → m a

class (Monoid w, Monad m) ⇒
  MonadWriter w m | m → w where
    tell :: w → m ()
```

Figure 3. Monads and monad transformer types.

the advice chain. The behavior of *proceed* for advice and base programs are, respectively, akin to *super* and *this* in an OO language.

In the Haskell approach presented in this section, the *proceed* argument to advice or base programs is always explicitly passed. However, it is possible to make *proceed* implicit using *implicit parameters* (Lewis et al. 2000).

EffectiveAdvice captures the essence of Aldrich's Open Modules. As in Open Modules, a programmer must anticipate the points at which advice can be applied by declaring a component *Open*. This is different from most aspect-oriented languages, which allow advice to be applied anywhere, at the cost of potentially breaking any advice when any part of a program changes. Having an explicit interface for advice places some burden on programmers, but has significant benefits in terms of modularity and modular reasoning.

2.2 Monads as Explicit Effects for EffectiveAdvice

For practical applications pure advice is of limited use. Most well-known examples of advice are effectful, including logging, tracing, backups, and memoization. A setting without effects is severely limited. Take the example in Section 2.1. Ideally it should be possible to construct a dynamic lookup table for the calls of the fibonacci function. However, without effects, the best we can do is to build in a static lookup table for some of the calls. Effectful advice is useful to provide a better solution for this problem, allowing the creation of a dynamic memo table where previously computed calls can be looked up.

A simple effectful memoization advice is presented in Figure 2. Figure 3 shows a number of monad and monad transformer (Liang et al. 1995) definitions that are used throughout the paper. The *MonadState* class, which models state, is used by the *memo* aspect to read and update the cached values in the memo table. The memo table is implemented using a map from integers to integers. If the input value to the function exists in the memo table, then the associated value is returned. Otherwise, the call proceeds and the memo table is updated with the input value and the result of the call.

The introduction of effects requires a change to the fibonacci component: it too must be written in a monadic manner, though it is fully parametric in the monad type. We can instantiate different monads, using the corresponding run functions in Figure 3, to recover variations of the fibonacci function. For example, the identity monad recovers the effect-free function

```
slowfib₂ :: Int → Int
slowfib₂ = runId ∘ weave fib₂
```

while a fast fibonacci function is obtained by adding the memo advice and suitably instantiating the state monad:

```
evalState :: State s a → s → a
evalState (State f) s = fst $ f s

fastfib :: Int → Int
fastfib n = evalState (weave (memo ⊕ fib₂) n) empty
```

Reasoning about the equivalence of EffectiveAdvice components does not require special-purpose mechanisms such as Aldrich's logical equivalence laws. Instead, Haskell's equational reasoning directly applies to effects modeled as monads. Consider the following two variants of *fib₂*

```
fib₃ proceed n = case n of
  0 → return 0
  1 → return 1
  _ → do y ← proceed (n − 2)
         x ← proceed (n − 1)
         return (x + y)

fib₄ proceed n = case n of
  0 → return 0
  1 → return 1
  _ → do m ← return (n − 1)
         x ← proceed m
         y ← proceed (m − 1)
         return (x + y)
```

Are these two variants indistinguishable with respect to *fib₂* for any client? Obviously it is not possible to show this for *fib₃*, which has switched the recursive calls, because it is possible, for example, to use tracing advice to notice that recursive calls are in a different order. However, straightforward equational reasoning shows that *fib₂* ≡ *fib₄*:

```
  do m ← return (n − 1)
     x ← proceed m
     y ← proceed (m − 1)
     return (x + y)
≡ {-Monad left unit: return x >>= f = f x -}
  do x ← proceed (n − 1)
     y ← proceed (n − 1 − 1)
```

```
data EvenOdd m = EO {
  even :: Int → m Bool,
  odd  :: Int → m Bool
}
evenodd :: Monad m ⇒ Open (EvenOdd m)
evenodd proceed = EO ev od where
  ev x | x ≡ 0      = return True
       | otherwise = odd proceed (x − 1)
  od x | x ≡ 0      = return False
       | otherwise = even proceed (x − 1)
logEO :: MonadWriter String m ⇒ Open (EvenOdd m)
logEO proceed = EO (log "even" (even proceed))
                   (log "odd" (odd proceed))
```

Figure 4. Mutually recursive definitions.

```
data Expr where
  Lit      :: Int → Expr
  Var      :: String → Expr
  Plus     :: Expr → Expr → Expr
  Minus    :: Expr → Expr → Expr
  Assign   :: String → Expr → Expr
  Sequence :: [Expr] → Expr
  While    :: Expr → Expr → Expr
type Env = [(String, Int)]
```

Figure 5. Datatype and environment type for expressions.

```
    return (x + y)
≡ {-n − 1 − 1 ≡ n − 2 -}
  do x ← proceed (n − 1)
     y ← proceed (n − 2)
     return (x + y)
```

The same approach shows that the two pure functions $slowfib_2$ and *fastfib*, or alternative implementations of the *memo* advice, are equivalent.

Mutual Recursion In EffectiveAdvice, mutual recursive functions can also be defined. In Haskell this is achieved using records, as shown in Figure 4. The record type *EvenOdd* is the signature for a pair of two functions *even* and *odd* and the component *evenodd* provides an implementation. The advice *logEO* adds logging to those definitions (the implementation of *log* is shown in Figure 7). Logged versions of *even* and *odd* are recovered through weaving as follows:

```
leven = runWriter ∘ even (weave (logEO ⊕ evenodd))
lodd  = runWriter ∘ odd (weave (logEO ⊕ evenodd))
```

Mutual recursion can also be used to introduce additional functions that can be advised. The designer of a component must anticipate where extensions maybe useful, although the designer need not predict what kind of extensions are made. All of advice, open and closed components fit in the same purely functional framework and abide by the same reasoning principles.

3. Advising Effectful Programs

There is a significant gap between the pure core language presented by Aldrich and realistic AOP systems like AspectJ. Kiczales and Mezini (2005) noted this gap and concluded that the restrictions in terms of expressiveness in such an approach may just be too limiting.

```
beval :: MonadState Env m ⇒ Open (Expr → m Int)
beval proceed exp = case exp of
  Lit x          → return x
  Var s          → do e ← get
                      case lookup s e of
                        Just x → return x
                        _      → error msg
  Plus l r       → do x ← proceed l
                      y ← proceed r
                      return (x + y)
  Minus l r      → do x ← proceed l
                      y ← proceed r
                      return (x − y)
  Assign x r     → do y ← proceed r
                      e ← get
                      put ((x, y) : e)
                      return y
  Sequence []    → return 0
  Sequence [x]   → proceed x
  Sequence (x : xs) → proceed x ≫ proceed (Sequence xs)
  While c b      → do x ← proceed c
                      if (x ≡ 0) then return 0
                        else (proceed b ≫ proceed exp)
  where msg = "Variable not found!"
```

Figure 6. A monadic evaluator using advice.

We disagree with this conclusion. In this section, we show how EffectiveAdvice scales to a setting which is much more expressive and realistic than Aldrich's Open Modules on two accounts:

1. it allows effects on both base programs and advice, and
2. it handles many kinds of effects, not just state.

In the remainder of this section we elaborate on these two points and illustrate them by implementing a modular monadic interpreter.

3.1 Effects for Base Programs

With EffectiveAdvice's monadic approach, both advice and base program may be effectful using monads. An example of an effectful base program is the monadic interpreter in Figure 6 for the simple imperative language of Figure 5. The interpreter's type *Open* (*Expr* → *m Int*) can be understood in the by now familiar way: it exports a function of type *Expr* → *m Int* and a join point of the same type. The type variable m means that advice may introduce effects. However, the constraint on m is now not *Monad m* for an unknown type of effect. Instead it is *MonadState Env m*: the effect must involve an updateable state of type *Env*, the environment used by the interpreter. In other words, the interpreter itself is effectful. In dealing with the *Var* and *Assign* cases it reads and writes the environment with the *get* and *put* functions.

A basic unadvised monadic evaluator is recovered as follows:

```
eval :: Expr → State Env Int
eval = weave beval
```

The exported join point is sealed and m is instantiated to the state monad.

3.2 Effects Beyond State

State is the only effect considered by Aldrich, but not the only useful effect that can be introduced by advice. In contrast, EffectiveAdvice allows any effect expressible as a monad, including output streams, exceptions, I/O, non-determinism, and combinations

```
log :: (MonadWriter String m, Show a, Show b)
   ⇒ String → Open (a → m b)
log name proceed x = do
   tell ("Entering " ++ name ++ "with" ++ show x ++ "\n")
   y ← proceed x
   tell ("Exiting " ++ name ++ "with" ++ show y ++ "\n")
   return y
```

Figure 7. The logging aspect.

```
dump :: (MonadState s m, MonadWriter String m, Show s)
     ⇒ Open (a → m b)
dump proceed arg =
  do s ← get
     tell (show s ++ "\n")
     proceed arg
```

Figure 8. The environment dumping aspect.

```
type Exc = (String, Expr, Env)
eeval :: (MonadState Env m, MonadError Exc m) ⇒
         Open (Expr → m Int)
eeval proceed exp = case exp of
   Var s → do e ← get
              case lookup s e of
                 Just x → return x
                 _      → throwError (msg, exp, e)
   _     → proceed exp
   where msg = "Variable not found!"
```

Figure 9. The exception handling aspect.

thereof. This point is illustrated next with three interesting uses of effect in advice.

Logging aspect Figure 7 shows how to define a logging aspect modularly. The advice writes a log message when entering the function call, delegates to *proceed* and finally writes another log message when exiting. It uses the writer monad transformer in Figure 3 for writing logging messages.

Dumping aspect Figure 8 shows how to define modular advice for dumping the environment at each evaluation step. The aspect intercepts the evaluation of every expression, retrieves the current environment, writes it out using a writer monad transformer and delegates the actual evaluation to *proceed*. This example is interesting because it shows that the advice not only introduces its own writer effect, but also relies on the presence of the state effect.

Exception handling aspect A last example of a useful aspect is given in Figure 9, to provide a better error handling facility for the interpreter. In the interpreter, an error can occur when a variable is looked up in the environment. The exception handling aspect overrides the case for variables and replaces the *error* primitive by *throwError* (see Figure 3). There are two advantages of using *throwError* instead of *error*. The first advantage is that additional useful information can be returned together with the exception (with *error* it is only possible to provide a string error message). For example, it may be useful to return the current environment, or the expression where the error has occurred so that the user can more easily identify the locale in the program that is to blame. The second advantage is that the exception is now explicit on the type of the evaluator and the client code must handle the exception, which ensures that the main program remains in a usable state. Like with the dumping aspect, two different types of monads are involved: a state and an error monad.

Weaving in functionality The different aspects can be combined in various ways, bringing together different effects or shared uses of the same effect:

```
debug1, debug2 :: (MonadWriter String m,
   MonadState Env m) ⇒ Expr → m Int
debug1 = weave (log "eval" ⊕ beval)
debug2 = weave (log "eval" ⊕ dump ⊕ beval)

exc :: (MonadError Exc m, MonadWriter String m,
        MonadState Env m) ⇒ Expr → m Int
exc = weave (eeval ⊕ log "eval" ⊕ beval)
```

The $debug_1$ program adds logging of function calls to the evaluator, while $debug_2$ is more verbose and also dumps the environment at each call. Finally, the third program logs calls, and may throw an exception if a variable that does not exist in the environment is used.

These programs can be run by picking suitable monads and extracting the relevant information. For example, in the programs shown next, the log string is returned (except if an error occurs).

```
getLog = snd ∘ fst
test1 e = getLog $
   runState (runWriterT (debug1 e)) []
test2 e = getLog $
   runState (runWriterT (debug2 e)) []
test3 e = extract $
   runStateT (runWriterT (exc e)) [] where
   extract (Left (msg, exp, _)) =
     "Error: " ++ msg ++
     "\nIn Expression: " ++ show exp
   extract (Right t) = getLog t
```

While the first two programs may silently give an error if a variable is not in the environment, the last program has to handle the exception explicitly and it can report an error message with the faulty expression.

4. Interference Combinators

Rinard et al. (2004) propose a classification system for interference patterns that can occur between advice and advised programs: *direct interference* consists of control flow manipulations, whereas *indirect interference* consist of state manipulations. They use program analysis to *identify* those patterns automatically.

EffectiveAdvice takes a different approach by providing combinators to *enforce* the different interference patterns at aspect composition time. Each combinator associates a particular type shape with an interference pattern. Thus, a composition that does not meet the type shape required by the combinator fails to type-check. Note that no special purpose extension of the type system is needed for this approach.

4.1 Enforcing Control Flow Properties

Direct interference is related to control flow and how the use of *proceed* calls can guarantee that a program satisfies certain properties. According to Rinard et al., advice can be classified as:

- **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*.

Consider the logging advice *log* of the previous section. This advice calls *proceed* exactly once. Therefore *log* is an example of augmentation advice. In EffectiveAdvice, the different forms of direct interference are enforced, rather than identified, using combinators. These interference combinators are discussed below.

Combination There is no new combinator since no interference properties are enforced. The $\oplus$ operator already composes advice of the general form $Open\ s$.

Replacement The informal requirement for replacement is that no calls are made to *proceed*. This requirement can be captured by the following combinator:

```
type Replace s = s

replace :: Replace s → Open s
replace radv = λproceed → radv
```

Replacement advice has type $Replace\ s$, which is the same type as the whole program. This reflects the fact that replacement advice is a proper program by itself. In other words the base program's behavior is replaced (or overridden) entirely, which has the effect of destroying the usual control flow of the base program.

Augmentation The informal requirement for augmentation advice is that *proceed* is called exactly once. This behavior is enforced with the *augment* combinator

```
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 (bef, aft) proceed a =
  do {c ← bef a; b ← proceed a; aft a b c; return b}
```

This combinator is responsible for calling *proceed* itself, rather than delegating this responsibility to the advice. The augmentation advice has type $Augment\ a\ b\ c\ m$, and it consists of two components: the first component is called *before proceed* and the second is called *afterwards*. Both parts can use the input a, but only the *after* argument has access to the result b of *proceed*. Moreover, the *before* part can communicate an auxiliary value c to the *after* part. For instance, log_1 is logging advice

```
log1 :: (MonadWriter String m, Show a, Show b)
     ⇒ String → Augment a b () m
log1 name = (bef, aft) where
  bef x     = write "Entering " x
  aft _ y _ = write "Exiting " y
  write a b = tell (a ++ name ++ show b ++ "\n")
```

such that $log \equiv augment \circ log_1$.

Combinators similar to the well-known AOP notions of *before* and *after* advice, can be implemented on top of *augment*:

```
before :: Monad m ⇒ (a → m ()) → Open (a → m b)
after  :: Monad m ⇒ (a → b → m ()) → Open (a → m b)
before bef =
  augment (λa → bef a ≫ return (), λa b c → return ())
after aft =
  augment (\_ → return (), λa b c → aft a b)
```

Our earlier dumping advice can be written as *before* advice:

```
dump1 :: (MonadState s m, MonadWriter String m, Show s)
      ⇒ a → m ()
dump1 arg =
  do s ← get
     tell (show s ++ "\n")
```

Note that $dump \equiv before\ dump_1$.

Narrowing This form of advice calls *proceed* at most once. Hence, a runtime choice can be made between replacement or augmentation advice:

```
type Narrow a b c m =
  (a → m Bool, Augment a b c m, Replace (a → m b))
narrow :: Monad m ⇒
  Narrow a b c m → Open (a → m b)
narrow (p, aug, rep) proceed x =
  do b ← p x
     if b then augment aug proceed x
          else replace rep proceed x
```

The runtime choice is made by the predicate of type $a \to m\ Bool$, i.e. based on the input a and monad m.

A typical example of narrowing is *memoization*. In the case of a repeated call, normal evaluation is *replaced* by a table lookup. In case of a new call, normal evaluation is *augmented* with tabulation.

```
memo1 :: (MonadState (Map a b) m, Ord a)
  ⇒ Narrow a b () m
memo1 = (p, (bef, aft), rep) where
  p x       = do {m ← get; return (member x m)}
  bef _     = return ()
  aft x r _ = do {m ← get; put (insert x r m)}
  rep x     = do {m ← get; return (m ! x)}
```

This version of memoization makes it clear that proceed is called at most once.

4.2 Enforcing Data Flow Properties

Indirect interference is related to data flow through the possible interaction of shared effects (or data) between advice and base programs. The most common form of shared effects is that of shared state. Another conventional form of effectful interaction is the throwing and catching of exceptions.

Rinard et al. (2004) consider five different forms of interference between advice and method (of the base program), specific to state:

- **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.

EffectiveAdvice generalizes these notions from state to arbitrary effects. Just as for control flow interference, it provides a number of combinators that enforce the form of effect interference.

Interference Primitives Interference arises by bringing together two programs, advice and a base program. EffectiveAdvice builds interference combinators from primitive combinators for individual programs. These primitives express whether the advice with effect t knows the type of effect m of the base program. If it does not know the type, then it cannot initiate interference. This absence of knowledge is captured by a higher-ranked type (Peyton Jones et al. 2007) and a corresponding conversion function to plain advice:

$$\begin{aligned}&\textbf{type } \mathit{NIAdvice}\ a\ b\ t = \forall m.(\mathit{Monad}\ m, \mathit{Monad}\ (t\ m))\\&\quad \Rightarrow \mathit{Open}\ (a \to t\ m\ b)\\&\mathit{niadvice} :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m))\\&\quad \Rightarrow \mathit{NIAdvice}\ a\ b\ t \to \mathit{Open}\ (a \to t\ m\ b)\\&\mathit{niadvice}\ \mathit{adv} = \mathit{adv}\end{aligned}$$

The opposite case does not require a new operator, since the plain type $\mathit{Open}\ (a \to t\ m\ b)$ suggests that interference may be possible.

Similarly, for the base program interference may not be initiated with:

$$\begin{aligned}&\textbf{type } \mathit{NIBase}\ a\ b\ m = \forall t.(\mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m))\\&\quad \Rightarrow \mathit{Open}\ (a \to t\ m\ b)\\&\mathit{nibase} :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m))\\&\quad \Rightarrow \mathit{NIBase}\ a\ b\ m \to \mathit{Open}\ (a \to t\ m\ b)\\&\mathit{nibase}\ \mathit{bse} = \mathit{bse}\end{aligned}$$

The types $\mathit{NIAdvice}$ and NIBase allow us to separate the effects that can be manipulated by the advice from the effects that can be manipulated by the base program. The type system guarantees that this is indeed the case.

In their general form the types of log_1 and beval are not sufficiently instantiated to establish non-interference. In fact, it is possible to obtain both interference and non-interference, depending on the instantiation of the monad.

Fortunately, the type checker confronts us with this issue by rejecting $\mathit{niadvice}\ \$\ \mathit{augment}\ \$\ \mathit{log}_1$ `"eval"` and $\mathit{nibase}\ \mathit{beval}$. The solution is to instantiate the types such that the overall effect monad is cleanly split into two independent parts, one for the advice and one for the base program:

$$\begin{aligned}&\mathit{log}_2 :: (\mathit{Show}\ a, \mathit{Show}\ b) \Rightarrow \mathit{NIAdvice}\ a\ b\ (\mathit{WriterT}\ \mathit{String})\\&\mathit{log}_2 = \mathit{augment}\ (\mathit{log}_1\ \texttt{"eval"})\end{aligned}$$

$$\begin{aligned}&\mathit{beval}_1 :: \mathit{NIBase}\ \mathit{Expr}\ \mathit{Int}\ (\mathit{State}\ \mathit{Env})\\&\mathit{beval}_1 = \mathit{beval}\end{aligned}$$

Interference Combinators Using the above primitives, EffectiveAdvice defines four primitive interference combinators:

$$\mathit{adv} \ominus \mathit{bse} = \mathit{niadvice}\ \mathit{adv} \oplus \mathit{nibase}\ \mathit{bse}$$

$$\mathit{adv} \circledcirc \mathit{bse} = \mathit{adv} \oplus \mathit{nibase}\ \mathit{bse}$$

$$\mathit{adv} \oslash \mathit{bse} = \mathit{niadvice}\ \mathit{adv} \oplus \mathit{bse}$$

$$\mathit{adv} \otimes \mathit{bse} = \mathit{adv} \oplus \mathit{bse}$$

Note that, unlike Rinard's categories, these combinators are not specific for state: they are parametric in the type of effect. The combinators $\otimes$ and $\ominus$ closely correspond to Rinard's interference and orthogonal categories. The $\circledcirc$ and $\oslash$ combinators indicate which of the two programs is aware of the other's effects, which are thus shared between the two programs.

For instance, the composition $\mathit{log}_2 \ominus \mathit{beval}_1$ expresses that the logging advice and the monadic evaluator do not interfere with each other's effects.

Stateful Effects Rinard et al. (2004) consider more refined forms of stateful interaction, based on read-only or read&write access to a shared state. EffectiveAdvice distinguishes between such forms of interaction by imposing appropriate constraints on the monad type variable m.

For this purpose EffectiveAdvice refines $\mathit{MonadState}$ to cater for different views:

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MGet}\ s\ m \mid m \to s\ \textbf{where}\\&\quad \mathit{get} :: m\ s\end{aligned}$$

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MPut}\ s\ m \mid m \to s\ \textbf{where}\\&\quad \mathit{put} :: s \to m\ ()\end{aligned}$$

$$\textbf{class } (\mathit{MGet}\ s\ m, \mathit{MPut}\ s\ m) \Rightarrow \mathit{MonadState}\ s\ m$$

The constraint $\mathit{MGet}\ s\ m$ only allows reading the state s of monad m, while the class MPut only allows writing it. The new $\mathit{MonadState}\ s\ m$ allows both reading and writing by subclassing both MGet and MPut. Four laws govern the semantics of the get and put methods:

$$\begin{aligned}\mathit{get} \gg m &\equiv m\\ \mathit{get} \gg\!= \lambda s_1 \to \mathit{get} \gg\!= f\ s &\equiv \mathit{get} \gg\!= \lambda s_1 \to f\ s\ s\\ \mathit{put}\ x \gg \mathit{put}\ y &\equiv \mathit{put}\ y\\ \mathit{put}\ x \gg \mathit{get} &\equiv \mathit{put}\ x \gg \mathit{return}\ x\end{aligned}$$

The new classes allow more accurate types, for instance dumping advice only requires reading the state:

$$\begin{aligned}&\mathit{dump}_2 :: (\mathit{MGet}\ s\ m, \mathit{MonadWriter}\ \mathit{String}\ m, \mathit{Show}\ s)\\&\quad \Rightarrow a \to m\ ()\\&\mathit{dump}_2\ _ = \textbf{do}\ \{s \leftarrow \mathit{get}; \mathit{tell}\ (\mathit{show}\ s +\!\!+ \texttt{"\textbackslash n"})\}\end{aligned}$$

With the two new constraints, EffectiveAdvice also defines relaxed versions of $\mathit{NIAdvice}$:

$$\textbf{type } \mathit{ROAdvice}\ a\ b\ t\ s\ = \forall m.(\mathit{MGet}\ s\ m, \mathit{MGet}\ s\ (t\ m)) \Rightarrow \mathit{Open}\ (a \to t\ m\ b)$$

$$\textbf{type } \mathit{WOAdvice}\ a\ b\ t\ s = \forall m.(\mathit{MPut}\ s\ m, \mathit{MPut}\ s\ (t\ m)) \Rightarrow \mathit{Open}\ (a \to t\ m\ b)$$

The dump_3 advice instantiates dump_2 as a $\mathit{ROAdvice}$:

$$\begin{aligned}&\mathit{dump}_3 :: \mathit{Show}\ s \Rightarrow \mathit{ROAdvice}\ a\ b\ (\mathit{WriterT}\ \mathit{String})\ s\\&\mathit{dump}_3 = \mathit{before}\ \mathit{dump}_2\end{aligned}$$

The new interference primitives in turn allow Rinard's state-specific interference classes to be expressed as combinators:

$$\begin{aligned}&\mathit{observation} :: (\mathit{MGet}\ s\ m, \mathit{MGet}\ s\ (t\ m), \mathit{MonadTrans}\ t) \Rightarrow\\&\quad \mathit{ROAdvice}\ a\ b\ t\ s \to \mathit{NIBase}\ a\ b\ m \to \mathit{Open}\ (a \to t\ m\ b)\\&\mathit{observation}\ \mathit{adv}\ \mathit{bse} = \mathit{adv} \oplus \mathit{bse}\end{aligned}$$

$$\begin{aligned}&\mathit{actuation} :: (\mathit{MPut}\ s\ m, \mathit{MPut}\ s\ (t\ m), \mathit{MonadTrans}\ t) \Rightarrow\\&\quad \mathit{WOAdvice}\ a\ b\ t\ s \to \mathit{NIBase}\ a\ b\ m \to \mathit{Open}\ (a \to t\ m\ b)\\&\mathit{actuation}\ \mathit{adv}\ \mathit{bse} = \mathit{adv} \oplus \mathit{bse}\end{aligned}$$

EffectiveAdvice puts similar constraints on the base program and distinguishes nine different forms of interference. Figure 10 connects these nine forms to the corresponding four terms used by Rinard et al.

Note that, by distinguishing between MPut and $\mathit{MonadState}$, EffectiveAdvice has a more fine-grained classification. $\mathit{MPut} \times \mathit{MPut}$, for instance, is only a weak form of interference. While both programs write to the same state, neither program's computations are affected; only the resulting state is.

While Rinard's classification is specific for state, EffectiveAdvice allows similar classifications for other kinds of effects. For example, with exceptions the rights to throw and catch exceptions are

	$MGet$	$MPut$	$MonadState$
$MGet$	Independent	Observation	Observation
$MPut$	Actuation	Interference	Interference
$MonadState$	Actuation	Interference	Interference

Figure 10. Refined state-based interference patterns (advice on the left, base program at the top).

separated into different monad subclasses: $MonadThrow\ e\ m$ for throwing an exception e, $MonadCatch\ e\ m$ for catching, and $MonadException\ e\ m$ for both. By considering the permitted operations of the advice and base program, the possible interference patterns between them are established.

5. Harmless Advice: Strong Guarantees of Non-Interference

This section uses direct and indirect non-interference combinators to enforce strong guarantees of non-interference.

5.1 Harmless Advice

The *harmless composition* combinator $\circledast$ ensures both control and data flow properties.

$$\textbf{type}\ NIAugment\ a\ b\ c\ t = \forall m.(Monad\ m, Monad\ (t\ m)) \Rightarrow Augment\ a\ b\ c\ (t\ m)$$

$$(\circledast) :: (Monad\ m, MonadTrans\ t, Monad\ (t\ m)) \Rightarrow NIAugment\ a\ b\ c\ t \rightarrow NIBase\ a\ b\ m \rightarrow Open\ (a \rightarrow t\ m\ b)$$
$$adv \circledast bse = augment\ adv \ominus bse$$

Harmless composition requires a special type of non-interfering augmentation advice, which is defined by $NIAugment$. It is important that the advice used by $\circledast$ is augmentation since, for instance, if an effectful base program could be called by advice twice, it could give different results than if called only once. This is because the result may depend on the effects of the base program. The $\ominus$ combinator used by $\circledast$ ensures that the advice and the base program have non-interfering effects.

Dantas and Walker (2006) introduced the notion of *harmless advice* for advice that guarantees full non-interference with the base program:

> *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.*

The full non-interference provided by the $\circledast$ combinator enforces that the advice is harmless. Let us cast the informal notion of harmlessness in a formal theorem:

Theorem 1 (Harmless Advice) *Consider a base program bse and advice adv with the types:*

$$bse :: \forall t.(MonadTrans\ t, Monad\ (t\ \kappa)) \Rightarrow Open\ (t\ \kappa\ \alpha)$$
$$adv :: \forall m.(Monad\ m, Monad\ (\tau\ m)) \Rightarrow Augment\ \alpha\ \beta\ \gamma\ (\tau\ m)$$

where κ is a monad and τ a monad transformer. If a function $proj :: \forall m, a.Monad\ m \Rightarrow \tau\ m\ a \rightarrow m\ a$ exists that satisfies the property:

$$proj \circ lift \equiv id$$

, then advice adv is harmless with respect to bse:

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

Informally, the theorem states that, if we ignore the effects introduced by the advice, the advised program is equivalent to the unadvised program. The role of the projection function $proj$ is to ignore the effects introduced by the advice. The required property $proj \circ lift \equiv id$ expresses the intuition that projection has no impact if there are no effects.

This theorem is proved in the appendix. Rather than looking into the details of the proof itself, it is more interesting to look into the techniques used by the proof: equational reasoning and parametricity.

Equational reasoning is the basic mechanism used in purely functional languages to reason about programs. Equational reasoning allows replacing a program for an equivalent one in any context, which leads to a simple algebraic style of proofs about programs like the one in Section 2.2. In impure languages equational reasoning does not generally hold, because a program may implicitly depend on the context of that program.

Parametricity (Wadler 1989) allows the derivation of theorems for a whole class of programs, only knowing their type. Voigtländer (2009) has recently shown how to extend the parametricity approach to type constructor classes such as $Monad$. This way we can derive theorems about effectful programs without knowing the particular effects used.

Parametricity in its simplest form only holds for total, i.e. fully defined and terminating, programs. If partial and non-terminating programs are also allowed, the advice may introduce non-termination and partiality. This is our counterpart of "may change the termination behavior" in Dantas's and Walker's definition.

5.2 Harmless Effects

In order to suit the Harmless Advice theorem, advice cannot introduce arbitrary effects. There must be a suitable projection function for ignoring the effects. Such projection functions do indeed exist for several state-related monad transformers.

WriterT For the $WriterT$ monad transformer we define the following projection function:

$$projW :: \forall w\ m\ a.(Monad\ m, Monoid\ w) \Rightarrow WriterT\ w\ m\ a \rightarrow m\ a$$
$$projW\ m = runWriterT\ m \gg\!= return \circ fst$$

It is indeed suitable:

Lemma 1 *The function $projW$ is a suitable function for the Harmless Advice theorem:*

$$projW \circ lift \equiv id$$

With the help of $projW$, the Harmless Advice theorem establishes that the logging advice is harmless:

$$proj \circ weave\ (log_2\ \texttt{"eval"} \circledast beval_1) \equiv runIdT \circ weave\ beval_1$$

StateT We can also define a suitable projection function for the $StateT$ monad transformer:

$$projS :: \forall s\ m\ a.Monad\ m \Rightarrow s \rightarrow StateT\ s\ m\ a \rightarrow m\ a$$
$$projS\ s0\ m = runStateT\ m\ s0 \gg\!= return \circ fst$$

Indeed, the required property holds:

Lemma 2 *The function $projS\ \ s0$ is a suitable function for the Harmless Advice theorem:*

$$projS\ s0 \circ lift \equiv id$$

for any $s0$.

The proofs for both lemmas are presented in the appendix.

Other Harmless Effects There are several other harmless effects, such as IdT with trivial projection function $runIdT$, $ReaderT$ and variations on these.

5.3 Harmful effects

An interesting aspect of our theorem is that harmless advice may not introduce arbitrary effects. Only those effects for which a suitable projection function $proj$ exists, may be used in harmless advice.

Consider again the $ErrorT\ e$ monad transformer of Figure 3. We can only partially define the projection function:

$$\begin{aligned}&projE :: \forall e\ m\ a.Monad\ m \Rightarrow ErrorT\ e\ m\ a \rightarrow m\ a\\&projE\ m = runErrorT\ m \gg\!= \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of}\\&\qquad Left\ e \rightarrow ???\\&\qquad Right\ x \rightarrow return\ x\end{aligned}$$

In the case of an error, we cannot produce a value. We could attempt to fix this issue by parametrizing $projE$ with a default value d:

$$\begin{aligned}&projE' :: \forall e\ m\ a.Monad\ m \Rightarrow a \rightarrow ErrorT\ e\ m\ a \rightarrow m\ a\\&projE'\ d\ m = runErrorT\ m \gg\!= \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of}\\&\quad Left\ e \rightarrow return\ d\\&\quad Right\ x \rightarrow return\ x\end{aligned}$$

but now $projE'\ d :: \forall e\ m.Monad\ m \Rightarrow ErrorT\ e\ m\ a \rightarrow m\ a$ fixes the type parameter a to the type of d, which is inappropriate.

Dantas and Walker mention that "Harmless advice may ... use I/O." However, undiscriminated use of I/O may definitely interfere with I/O in the base program. In Haskell, this manifests itself in the fact that there is no safe way to project from the IO monad. Only more disciplined effects, such as $WriterT$, $ReaderT$ and $StateT$ are possible.

5.4 Harmless Observation Advice

In the main Harmless Advice theorem, we have used the $\circledast$ operator which enforces that advice and base program are orthogonal. While orthogonality is a sufficient condition, it is certainly not a necessary one. For instance, observation advice may be harmless too. A combinator that forces harmless observation advice is:

$$\begin{aligned}&\mathbf{type}\ NIOAugment\ a\ b\ c\ s\ t = \forall m.\\&\quad (MGet\ s\ m, Monad\ (t\ m)) \Rightarrow Augment\ a\ b\ c\ (t\ m)\\&(\odot) :: (MGet\ s\ m, MonadTrans\ t, MGet\ s\ (t\ m)) \Rightarrow\\&\quad NIOAugment\ a\ b\ c\ s\ t \rightarrow NIBase\ a\ b\ m \rightarrow Open\ (a \rightarrow t\ m\ b)\\&adv \odot bse = augment\ adv\ `observation`\ bse\end{aligned}$$

Now we can adapt the theorem accordingly:

Theorem 2 (Harmless Observation Advice) *Consider a base program* $bse :: \forall t.MonadTrans\ t \Rightarrow Open\ (t\ \kappa\ \alpha)$ *and advice* $adv :: \forall m.MGet\ \sigma\ m \Rightarrow Augment\ \alpha\ \beta\ \gamma\ (\tau\ m)$*, with* κ *a* $MonadState\ \sigma$ *and* τ *a* $MonadTrans$*. If a function* $proj :: \forall m\ a.Monad\ m \Rightarrow \tau\ m\ a \rightarrow m\ a$ *exists that satisfies the property:*

$$proj \circ lift \equiv id$$

, then the advice adv *is harmless with respect to* bse*:*

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

We refer to the appendix again for the proof. It is similar in style to that of the Harmless Advice theorem. The main difference lies in the fact that the advice knows more about the m type parameter. As a consequence, weaker parametricity results are obtained. The loss in parametricity is made up for by exploiting the two get laws.

Theorem 2 establishes that dumping advice is harmless:

$$projW \circ weave\ (dump_3 \odot beval_1) \equiv runIdT \circ weave\ beval_1$$

6. Future Work

This section discusses a few ideas for future work on EffectiveAdvice, including how to solve some of the current limitations.

Managing the monadic stack EffectiveAdvice relies on monad transformers to stack up the effects used by different concerns. This is very flexible because it allows the stack to grow as new concerns with additional effects are added. Additionally, (constrained) parametric polymorphism allows effects that are irrelevant to the concern in question to be ignored. This means that concerns are highly adaptable (since they work for monadic stacks of many different forms) and yet self-contained (they do not need to be modified after the addition of new concerns). However, we identify two issues with this monad stack approach:

1. Initializing and running the layers of the monad stack, while trivial for small stack, becomes more troublesome as the stack grows due to the big types involved.
2. The current monad transformer library (MTL) technology (Liang et al. 1995) makes it difficult to use two distinct monad transformers of the same kind in the same program. For example, we may want to keep separate logs for debugging and profiling.

A solution to the first problem, but in the context of a stack of *applicative functors* (McBride and Paterson 2008), was presented by Gibbons and Oliveira (2009): a type class for automatically initializing and running the stack. A similar technique could be applied to EffectiveAdvice, which would greatly alleviate the problems of having big types. Alternatively, combinators could be defined for managing the initialization of the monad stack. For the second problem, the *Monatron* monad transformer library (Jaskelioff 2008) poses a viable alternative. This library solves many of the issues of the MTL, including the problem of having monad transformers of the same kind in the same program.

Object-Oriented Languages The EffectiveAdvice methodology is not limited to Haskell or functional programming languages, but also applies to object-oriented languages like Scala. Haskell was chosen as the most established member of pure functional programming languages, which forces explicit effects and supports equational reasoning and parametricity. Without these properties, the theoretical developments presented in Section 5 regarding interference are not enforceable by the language. However, disciplined use of the presented combinators already brings many advantages.

Moreover, object-oriented languages, such as Scala, already provide valuable language-level support for scaling EffectiveAdvice features:

- Scala supports mixins natively. Hence, Scala classes are open to mixins by default, and the native support to inheritance avoids the explicit parametrization of arguments like $proceed$. This provides some convenience that the Haskell approach does not have.
- Grouping multiple functions in a class is directly supported by the language through objects, which can ultimately be viewed as groups of possibly mutually recursive functions. Hence, extending an open module from a single to multiple functions does not incur any notational overhead.

Also, subtyping poses an interesting alternative to type classes for expressing restricted rights to explicit effects. Furthermore, as

```
trait fibState[S] extends (Int ⇒ State[S, Int]) {
    def apply (n : Int) = n match {
      case 0 ⇒ 0
      case 1 ⇒ 1
      case _ ⇒ for (r1 ← this (n − 1);
                    r2 ← this (n − 2)) yield r1 + r2
    }
}
trait memoState
  extends (Int ⇒ State[HashMap[Int, Int], Int]) {
    abstract override def apply (arg : Int) =
      for (cache ← get;
           res ← (for (res ← cache.get (arg))
                    yield result[HashMap[Int, Int], Int] (res)
                  ) getOrElse (
                    for (res   ← super.apply (arg);
                         map ← get;
                         _     ← put (map.update (arg, res))
                    ) yield res
          )) yield res
}
object fmemo
    extends fibState[HashMap[Int, Int]] with memoState
def test (x : Int) =
    fmemo (x).runState (new HashMap[Int, Int] ())
```

Figure 11. Purely functional memoization in Scala.

shown by Moors et al. (2008), Scala supports all the necessary features to implement monads in the same way as Haskell.

Figure 11 illustrates the native support for mixins in action for a purely functional memoization example like the one presented in Section 2. The main points of the Scala solution are briefly discussed next. It is assumed that the code for the *State* monad is defined. The *fibState* component defines the open component for the fibonacci function. Noteworthy is the use of **this** instead of *proceed* and the **for** notation instead of the **do** notation. The *memoState* component makes use of mixin inheritance through an **abstract override** definition and a **super** call to override the *apply* method of the super class. The *fmemo* object plays the role of the weaver, combining the functionality of *fibState* and *memoState* together. Finally, *test* shows how to write a client.

In the short term, we would like to explore Scala's support for EffectiveAdvice further and compare it to the Haskell approach. In the longer term, we would like to develop a purely functional object-oriented language that combines the advantages of Haskell and Scala, namely strong reasoning properties and native support for mixins and objects.

Pointcuts Pointcut declarations allow the definition of sets of join points. This is useful to advise multiple join points with a single declaration, which allows easy deployment of *massively crosscutting concerns* such as logging. Typically advice and pointcut declarations are combined together, allowing statements such as "advise all the methods called *get* in the system", which are highly syntax-oriented. Our approach avoids such syntactic quantification and relies on explicit composition of aspects and programs. However, for massively cross-cutting concerns, a lot of compositions are required. We view this as the biggest limitation of EffectiveAdvice from a practical point of view. Finding a more semantic alternative to syntactic quantification, while avoiding the caveats of that mechanism is something that we hope to investigate in the future.

7. Related Work

Kiczales et al. (1997) introduced AOP and stated its goal: to modularize concerns that cut across the components of a software system. A more direct definition of AOP is proposed by Filman and Friedman (2000): the distinguishing characteristics of AOP systems are support for *quantification* and *obliviousness*. Quantification is the ability to write separate pieces of code that affect many different (non-local) places in a software system. Obliviousness means that the places affected by quantifications do not need to prepare for the additional behavior. This definition of AOP is broad and general enough to include related technologies, including feature-oriented programming, which might not be included in a narrow definition of AOP. Filman and Friedman also distinguish between *black-box* and *clear-box* AOP. In black-box AOP systems the quantification is over the (public) interface of components, whereas in clear-box AOP systems the quantification is over the parsed structure of components.

Clear-box AOP The original implementations of AOP (Kiczales et al. 1997) rely on *aspect languages* to advise programs written in *component languages* by using the names of entities present in the component programs. Typical implementations of AOP based on the *pointcut-advice model* (Wand et al. 2004), such as AspectJ (Eclipse-Foundation 2000-2009), use *pointcuts* to designate which set of *join points* to crosscut using the names of *classes*, *methods*, *modules* or any other entity containing code; and *advice* to specify what happens when a pointcut is reached. *Subject-oriented programming* (Harrison and Ossher 1993) has composition rules that allow quantifying over elements such as the interpretation of variables within modules. These approaches are examples of clear-box AOP, since they rely on knowledge about the internal structure of programs. Clear-box systems are usually very powerful and they allow easy deployment of massively crosscutting concerns, however they make modular reasoning very hard.

Functional AOP systems Two main approaches to functional AOP exist, both following the pointcut-advice model: 1) *statically typed language-based* approaches such as Aspectual Caml (Masuhara et al. 2005), AspectFun (Chen et al. 2007) and AspectML (Dantas et al. 2008), and 2) *lightweight dynamically typed* approaches such as AspectScheme (Dutchyn et al. 2006). While the statically typed approach has obvious benefits, dynamically typed languages usually allow more lightweight library-based solutions. This has benefits in terms of *reusable aspects* (Fraine and Braem 2007) and expressing *dynamically deployed* aspects (Tanter 2008). In some sense, EffectiveAdvice combines the best of both worlds: it is a very lightweight *statically typed library-based* approach. However, it uses a model of explicit composition of advice instead of the pointcut model. In EffectiveAdvice, a number of "features" (such as *first-class*, *polymorphic* and *inferable types for* advice) come essentially for free. In language-based approaches adding support for each of these features is non-trivial, and only AspectML supports all of them.

Black-box AOP Filman and Friedman note that many existing mechanisms in programming languages already support some defining characteristics of AOP. For example, *inheritance* in conventional object-oriented languages supports a limited form of quantification since code in a super class has an effect on all its subclasses. *Feature-oriented programming* (FOP) (Prehofer 1997) proposes a model for object-oriented programming that generalizes inheritance but, unlike inheritance mechanisms found in most mainstream languages, features are composed at run-time. Consequently, FOP has greater potential for reusable and dynamically

deployed aspects. These approaches are examples of black-box AOP: they quantify over the public interfaces of components. As a consequence, they tend to have better reasoning properties than clear-box approaches. Apel et al. (2007) illustrate this with their formalization and exploitation of the mathematical structure behind FOP.

Mixins Filman and Friedman argue that many systems supporting a form of *mixin inheritance* (Cook 1989; Bracha and Cook 1990) have *oblivious quantification*, since the derived classes are unaware of the specific super classes that affect them. Thus, for them, mixin inheritance is a *full-blown* form of (black-box) AOP. Many authors have used mixin inheritance in functional programming (Cook 1989; McAdam 1997; Garrigue 2000; Läufer 2003; Brown and Cook 2007), using techniques similar to that presented in Section 2. However, only Brown and Cook (2007) have used it in a pure functional language with explicit effects, to modularize memoization.

Modular Reasoning Kiczales and Mezini (2005) argue that modular reasoning about cross-cutting aspects is not possible. Instead they propose a global analysis that infers interfaces of deployed systems. Changing one component may lead to pervasive changes of interfaces.

In contrast, Aldrich (2005) does define the concept of Open Modules that allows modular reasoning. However, this approach is severely limited: reasoning of equivalence is limited to pure base programs with respect to impure advice. Reasoning about effectful base programs or advice is not covered. Moreover, it is not clear at all what forms of effect are allowed in advice because the advice language is not part of the formal framework.

Interference Many authors have identified (non-)interference as an important factor in reasoning about advice.

Clifton and Leavens (2002) identify that observers (harmless observation advice) do not change the specification of any module they advise. However, Clifton and Leavens do not provide an approach for establishing whether an advice is an observer or not.

Rinard et al. (2004) formulate a classification scheme for different forms of interference, and combine a number of program analyses for automated classification. No formal results are proven.

Douence et al. (2004) present an approach for determining strong independence of stateful aspects: when aspects commute, they do not interfere with each other. Equational reasoning laws are used on aspect bodies to determine commutation; in contrast, EffectiveAdvice only looks at the types. While this paper focuses on the interaction of an aspect with a base program, the same approach applies equally to the interaction of two aspects.

Dantas and Walker (2006) propose a new type-and-effect system for identifying harmless advice on a core language: protection domains prevent information flow from advice to base program.

Clifton et al. (2007) propose an extension to AspectJ, which allows the declaration of *control* and *heap* effects. Control effects are related to control flow perturbations like not calling *proceed* or calling it multiple times and they also include exceptions. Heap effects track effects that affect the heap, such as assignments to fields.

In summary, existing approaches to non-interference formulate special-purpose program analyses or type systems. A major advantage of EffectiveAdvice over all of these is its extremely lightweight nature. Everything is built on top of existing and familiar language features; no new analysis or type system is required. Moreover, it is possible to reason formally about programs using familiar techniques such as equational reasoning and parametricity.

Aspects and Effects The connection between AOP and effects is a recurring theme of discussion since De Meuter (1997) argued about the use of monads as a theoretical foundation for AOP. But this view is not widely accepted. Hofer and Ostermann (2007) argued recently that "*monads and aspects have to be regarded as quite different mechanisms*". EffectiveAdvice shows that aspects (when understood as advice) and effects have complementary roles when it comes to separation of concerns: aspects provide textual separation of code, and effects provide conceptual separation of effects used by different aspects. The relationship between aspects and effects is not one-to-one, since an aspect may produce several types of effects, and the same effect may be manipulated by several different aspects.

8. Conclusion

EffectiveAdvice promotes the idea that effects should be an integral part of the interface of components, avoiding hidden data flows between components. This has important benefits:

- Modular reasoning is possible, since only the implementation of a program and the interfaces of the components used by that program are needed to understand that program locally.
- Reasoning about the interference between components becomes possible by looking at the interfaces of components.

The EffectiveAdvice methodology can be implemented in existing programming languages. In this paper a very simple and lightweight implementation is presented as a Haskell library. The methodology can also be implemented in a language like Scala, although implicit data flows cannot be ruled out.

Acknowledgments

We are grateful to Benjamin Delaware, Marko van Dooren, Jeremy Gibbons, Simon Peyton Jones, Shriram Krishnamurthi, Adriaan Moors, Janis Voigtländer and Meng Wang for their useful comments; to Jonathan Aldrich for valuable feedback on Open Modules; and to Andres Löh for supporting `lhs2tex`.

References

Jonathan Aldrich. Open modules: Modular reasoning about advice. In *LNCS 3586: European Conference on Object-Oriented Programming*, pages 144–168, 2005.

Sven Apel, Christian Lengauer, Don Batory, Bernhard Möller, and Christian Kästner. An algebra for feature-oriented software development. Technical report, University of Passau, 2007.

Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA '90*, pages 303–311, 1990.

Daniel Brown and William R. Cook. Monadic memoization mixins. Technical Report TR-07-11, The University of Texas, 2007.

Kung Chen, Shu-Chun Weng, Meng Wang, Siau-Cheng Khoo, and Chung-Hsin Chen. A compilation model for aspect-oriented polymorphically typed functional languages. In *International Symposium on Static Analysis*, volume 4634 of *LNCS*, pages 34–51, 2007.

Curtis Clifton and Gary T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *Foundations of Aspect Languages*, pages 33–44, 2002.

Curtis Clifton, Gary T. Leavens, and James Noble. MAO: Ownership and effects for more effective reasoning about aspects. In *European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 451–475, 2007.

William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

Daniel S. Dantas and David Walker. Harmless advice. In *POPL '06: Conference record of the 33rd -SIGACT symposium on Principles of programming languages*, pages 383–396, 2006.

Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. Aspectml: A polymorphic aspect-oriented functional programming language. *ACM Trans. Program. Lang. Syst.*, 30(3):1–60, 2008.

Wolfgang De Meuter. Monads as a theoretical foundation for AOP. In *International Workshop on Aspect-Oriented Programming at ECOOP.*, 1997.

Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: International conference on Aspect-oriented software development*, pages 141–150, 2004.

Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program.*, 63(3):207–239, 2006.

Eclipse-Foundation. Aspectj, 2000-2009. See `http://eclipse.org/aspectj/`.

Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns*, pages 21–35, 2000.

Bruno De Fraine and Mathieu Braem. Requirements for reusable aspect deployment. In *Software Composition*, pages 176–183, 2007.

Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, 2000.

Jeremy Gibbons and Bruno Oliveira. The essence of the Iterator pattern. In *Journal of Functional Programming*, volume 19, pages 377–402, 2009.

William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. *Not.*, 28(10):411–428, 1993.

Christian Hofer and Klaus Ostermann. On the relation of aspects and monads. In *FOAL '07: Foundations of aspect-oriented languages*, pages 27–33, 2007.

Mauro Jaskelioff. Monatron: An Extensible Monad Transformer Library. In *IFL 2008*, 2008.

Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Programming Languages and Systems*, pages 230–244, 2000.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. 2003.

Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ACM: International Conference on Software engineering*, pages 49–58, 2005.

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *LNCS 1241: European Conference on Object-Oriented Programming*, pages 220–242, 1997.

Konstantin Läufer. What functional programmers can learn from the Visitor pattern. Technical report, Loyola University Chicago, 2003.

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: dynamic scoping with static types. In *POPL '00: Symposium on Principles of Programming Languages*, pages 108–118, 2000.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages*, pages 333–343, 1995.

Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, 2006. ISBN 1-59593-196-1.

Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proc. of the tenth International Conference on Functional Programming*, pages 320–330, 2005.

Bruce J. McAdam. That about wraps it up — using fix to handle errors without exceptions, and other programming tricks. Technical report, Laboratory for Foundations of Computer Science, The University of Edinburgh, 1997.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2008.

Eugenio Moggi. Computational lambda-calculus and monads. In *LICS '89: Logic in Computer Science*, pages 14–23, 1989.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *OOPSLA '08: Object-oriented programming systems languages and applications*, pages 423–438, 2008.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(01):1–82, 2007.

Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97: European Conference on Object-Oriented Programming*, 1997.

Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. *ACM SIGSOFT Softw. Eng. Notes*, 29(6):147–158, 2004.

Raymie Stata and John V. Guttag. Modular reasoning in the presence of subclassing. *SIGPLAN Not.*, 30(10):200–214, 1995.

Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *AOSD '08: Aspect-oriented software development*, pages 168–179, 2008.

Janis Voigtländer. Free theorems involving type constructor classes. In *ICFP '09: International Conference on Functional programming*, 2009.

Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. 1989.

Philip Wadler. The essence of functional programming. In *POPL '92: Principles of Programming Languages*, pages 1–14, 1992.

Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

A. Background

This section provides a quick review of monads and monad transformers in Haskell. For a more detailed discussion see Liang et al. (1995). We assume familiarity with Haskell and type classes Jones (2003).

A.1 Monads

Monads are a standard technique for encapsulating computational effects in pure functional languages (Wadler 1992; Moggi 1989, 1991). Examples of computational effects include mutable state, error handling, and non-determinism. Monads allow explicit representation of *computations*, which produce values of a given type and may perform side effects. Computations are composed by a *bind* operator that hides the details of the computation effect (passing explicit state, handling errors, etc). In Haskell, monads are described by a type class:

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

The type function m describes computations of type $m\ a$ which produce values of type a when executed. The function *return* lifts a value of type a into a (pure) computation that simply produces the value. The *bind* function $\gg\!=$ composes a computation $m\ a$, which produces values of type a, with a function that accepts a value of type a and returns a computation of type b.

All instances of *Monad* must satisfy the following laws:

```
return x >>= f  = f x                          {-left unit -}
p >>= return    = p                            {-right unit -}
(p >>= f) >>= g = p >>= λx → (f x >>= g)       {-associativity -}
```

The Haskell **do** notation is syntactic sugar for the *bind* operator: **do** $\{x \leftarrow f; g\}$ means $f \gg\!= \lambda x \rightarrow g$. For example, a monadic evaluator for a simple type of expressions supporting only integer literals and division, can be written as follows:

```
data Term = Con Int | Div Term Term

meval1 :: Monad m ⇒ Term → m Int
meval1 (Con a)   = return a
meval1 (Div t u) = do x ← meval1 t
                      y ← meval1 u
                      return (x `div` y)
```

For integer terms, the integer denoted by the term is returned; for divisions, the value of the dividend is computed and bound to x, then the value of the divisor is computed and bound to y, and finally x `` `div` `` y is returned. The evaluator is parameterized by a monad to allow different computational effects to be introduced into the evaluation process.

A simple evaluator with no computational effects is defined by parameterizing $meval_1$ by the *identity monad*. In the identity monad, *return* is the identify function and $\gg\!=$ is reverse application (modulo the isomorphisms into and out of the *Id* type constructor):

```
newtype Id a = Id {runId :: a}

instance Monad Id where
  return x = Id x
  x >>= f  = f $ runId x
```

Where $ is right-associative low-precedence application. Using *Id*, the simple evaluator can can be defined as follows:

```
eval :: Term → Int
eval = runId ∘ meval1
```

A.2 Monad Transformers

A *monad transformer* (Liang et al. 1995) is a higher-order monad that is parameterized by another monad. Monad transformers are needed because monads do not compose well on their own. With monad transformers, different kinds of monads can be layered on top of each other to compose the functionality provided by each monad. A monad transformer is defined by the following type class:

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

The *lift* operation takes a monadic computation $m\ a$, and lifts it into the transformed monad $t\ m$. The most trivial example of monad transformers is given by the identity transformer, which performs no effects:

```
newtype IdT m a = IdT {runIdT :: m a}

instance Monad m ⇒ Monad (IdT m) where
  return  = IdT ∘ return
  p >>= k = IdT $ runIdT p >>= runIdT ∘ k

instance MonadTrans IdT where
  lift    = IdT
```

A more insteresting example is provided by the state monad transformer, defined in Figure 12. The new type $StateT\ s\ m\ a$ represents a computation with state of type s that incorporates the effects of m. The instance *MonadTrans* for $StateT\ s$ runs the inner computation and acts as an identity function on the state s. The $MonadState\ s\ m$ type class defines operation *get* and *put* to access and update the encapsulated state. An instance of that type class is defined for the state monad transformer type. Finally, an instance of *Monad* for $StateT\ s\ m$ is also needed. Note that *MonadState* requires *multiple parameter type classes with functional dependencies* (Jones 2000); both *MonadState* and *MonadTrans* as well as many other monad transformers can be found in the *Glasgow Haskell Compiler* (GHC) *monad transformer library*. Additional monad transformers from this library are used in this paper as needed, with only the relevant definitions presented. Details may be found in (Liang et al. 1995).

```
newtype StateT s m a = StateT {
  runStateT :: s → m (a, s)
}
instance MonadTrans (StateT s) where
    lift p = StateT $ λs → do {a ← p; return (a, s)}
instance Monad m ⇒ Monad (StateT s m) where
    return a = StateT $ λs → return (a, s)
    p >>= k = StateT $ λs → do
      (a, s') ← runStateT p s
      runStateT (k a) s'
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
instance (Monad m) ⇒
  MonadState s (StateT s m) where
    get = StateT $ λs → return (s, s)
    put s = StateT $ \_ → return ((), s)
```

Figure 12. State monad transformer machinery.

Using monad transformers it is possible to combine monads. Consider combining a state monad with an error monad, which has the following interface, where e is the type of error values:

```
newtype ErrorT e m a = ErrorT {
  runErrorT :: m (Either e a)
}
class Monad m ⇒ MonadError e m | m → e where
```

$$throwError :: e \to m\ a$$
$$catchError :: m\ a \to (e \to m\ a) \to m\ a$$

Using $MonadState$ and $MonadError$ it is possible to define a program that checks for division by 0 errors and counts the number of division performed:

```
meval₂ :: (MonadState Int m, MonadError String m) ⇒
          Term → m Int
meval₂ (Con a)   = return a
meval₂ (Div t u) =
  do x ← meval₂ t
     y ← meval₂ u
     n ← get
     put (n + 1)
     if y ≡ 0 then throwError "divide by zero!"
              else return (x `div` y)
```

In this program both the state monad and the error monad operations are used, so the monad m needs to be both an instance of $MonadState$ and $MonadError$.

The combination of the two monads must specify how stateful computations interact with errors. The goal is a monad m that is both a $MonadState\ s$ with get/put and a $MonadError\ e$ with $catch/throw$. Two solutions exist: either $ErrorT\ e\ (StateT\ s\ Id)$, which produces either a value or an error in an updated state, or $StateT\ s\ (ErrorT\ e\ Id)$, which either produces an error or a value and an updated state. Additional instances are needed to ensure that these compositions implement both $MonadState$ and $MonadError$, but the necessary definitions are all part of the standard monad library.

B. Proof of Harmlessness Theorem

In orther to show that the advice is truly harmless, we first prove a number of auxiliary lemmas.

First, we show how to convert between the self-explanatory form of augmentation advice used in the paper and the more dense form $a \to t\ m\ (b \to t\ m\ c)$ that is convenient for writing proofs.

The connection between the two forms is captured by the *convert* function, which translates from the former to the latter.

```
convert :: (Monad m, MonadTrans t, Monad (t m))
   ⇒ (a → t m c, a → b → c → t m ())
   → (a → t m (b → t m ()))
convert (bef, aft) =
  λa → bef a >>= (λc → return (λb → aft a b c))
```

The counterpart of the *augment* function is

```
around :: (Monad m, MonadTrans t, Monad (t m))
   ⇒ (a → t m (b → t m ()))
   → Open (a → t m b)
around adv = λproceed →
  λa → adv a >>= λaft →
    proceed a >>= λr →
    aft r     >>= \_ →
    return r
```

Lemma 3 *Consider augmentation advice* $(bef, aft) :: (\alpha \to \tau\ \kappa\ \gamma, \alpha \to \beta \to \gamma \to \tau\ \kappa\ ())$*, then we have that:*

$$augment\ (bef, aft) \equiv around\ (convert\ (bef, aft))$$

where κ is a $Monad$ and τ is a $MonadTrans$.

Proof:

$$\begin{aligned}
&around\ (convert\ (bef, aft)) \\
\equiv\ &\{\text{-unfold } around\ \text{-}\} \\
&(\lambda proceed \to \lambda a \to convert\ (bef, aft)\ a \gg\!= \lambda aft' \\
&\quad \to proceed\ a \gg\!= \lambda b \\
&\quad \to aft'\ b \gg\!= \backslash_ \\
&\quad \to return\ b) \\
\equiv\ &\{\text{-unfold } convert\ \text{-}\} \\
&(\lambda proceed \to \lambda a \to bef\ a \gg\!= \lambda c \\
&\quad \to return\ (\lambda b \to aft\ a\ b\ c) \gg\!= \lambda aft' \\
&\quad \to proceed\ a \gg\!= \lambda b \\
&\quad \to aft'\ b \gg\!= \backslash_ \\
&\quad \to return\ b) \\
\equiv\ &\{\text{-Monad left unit }\text{-}\} \\
&(\lambda proceed \to \lambda a \to bef\ a \gg\!= \lambda c \\
&\quad \to proceed\ a \gg\!= \lambda b \\
&\quad \to aft\ a\ b\ c \gg\!= \backslash_ \\
&\quad \to return\ b) \\
\equiv\ &\{\text{-fold } augment\ \text{-}\} \\
&augment\ (bef, aft)
\end{aligned}$$

□

Lemma 4 *Consider a function* $f :: \forall t.MonadTrans\ t \Rightarrow (\alpha \to t\ \kappa\ (\beta \to t\ \kappa\ ())) \to \alpha \to t\ \kappa\ \beta$ *with* κ *an arbitrary monad, then we have that:*

$$out\ \pi \circ f \equiv out\ unIdT \circ f \circ out\ (IdT \circ fmap\ (out\ (IdT \circ \pi)) \circ \pi)$$

for any $\pi :: \forall m, a.Monad\ m \Rightarrow \tau\ m\ a \to m\ a$ *with* τ *an arbitrary monad transformer that satisfies the following property:*

$$\pi \circ lift \equiv id$$

where $out = (\circ)$ *applies a function to the output of another function.*

Proof: Let $\mathcal{T} : \tau \Leftrightarrow IdT$ be the $MonadTrans$ action

$$\mathcal{T}\ \mathcal{F}\ \mathcal{R} = \pi; \mathcal{F}\ \mathcal{R}; IdT$$

. This is a $MonadTrans$ action indeed:

- $(lift_\tau, lift_{IdT}) \in \forall \mathcal{F}, \mathcal{R}.\mathcal{F}\ \mathcal{R} \to \mathcal{T}\ \mathcal{F}\ \mathcal{R}$, since for every $(a, b) \in \mathcal{F}\ \mathcal{R}$ we have $(lift_\tau\ a, lift_{IdT}\ b) = (lift_\tau\ a, IdT\ b) \in \pi; \mathcal{F}\ \mathcal{R}; IdT =$ because of Property (1) of π.

Then we have for all $(h, h') \in (id_\beta \to \mathcal{T}\ \kappa\ id_{()})$, that $h' \equiv IdT \circ \pi \circ h)$, because $\kappa\ id \equiv id$. Assume that $(g, g') \in id_\alpha \to \mathcal{T}\ \kappa\ (id_\beta \to \mathcal{T}\ \kappa\ id_{()})$, where $g' \equiv out\ (IdT \circ fmap\ (out\ (IdT \circ \pi)) \circ \pi)\ g$. Then, for $(f\ g, f\ g') \in id_\alpha \to \mathcal{T}\ \kappa\ id_\beta$ the lemma follows.

Now, we only have to show that the assumption wrt. (g, g') is valid. The assumption is valid if for all $(a, a) \in id_\alpha$, we have that $(ga, g'a) \in \mathcal{T}\ \kappa\ (id_\beta \to \mathcal{T}\ \kappa\ id_{()})$. This holds if, applying $\mathcal{T}$, we have that $(proj(ga), runIdT(g'a)) \in \kappa\ (id_\beta \to \mathcal{T}\ \kappa\ id_{()})$. By equational reasoning, we get

$$\begin{aligned}
&(\pi\ (g\ a), runIdT\ (g'\ a)) \\
\equiv\ &\{\text{-unfold } g'\ \text{-}\} \\
&(\pi\ (g\ a), runIdT \circ IdT \circ fmap\ (out\ (IdT \circ \pi))\ \$\ \pi\ (g\ a)) \\
\equiv\ &\{\text{-}runIdT \circ IdT \equiv id\ \text{-}\} \\
&(\pi\ (g\ a), fmap\ (out\ (IdT \circ \pi))\ \$\ \pi\ (g\ a)) \\
\equiv\ &\{\text{-unfold } fmap \text{ and } out\ \text{-}\} \\
&(\pi\ (g\ a), \pi\ (g\ a) \gg\!= \lambda f \to return\ (IdT \circ \pi \circ f)) \\
\equiv\ &\{\text{-Monad right unit }\text{-}\} \\
&(\pi\ (g\ a) \gg\!= return, \\
&\quad \pi\ (g\ a) \gg\!= \lambda f \to return\ (IdT \circ \pi \circ f))
\end{aligned}$$

Note that $(\pi\ (g\ a), \pi\ (g\ a)) \in \kappa\ \mathcal{R}$, and $(\gg\!=, \gg\!=) \in \kappa\ \mathcal{R} \to (\mathcal{R} \to \kappa\ \mathcal{S}) \to \kappa\ \mathcal{S}$, where $\mathcal{R} = id_\beta \to \tau\ \kappa\ id_{()} = id_{\beta \to \tau\ \kappa\ ()}$ and $\mathcal{S} = id_\beta \to \mathcal{T}\ \kappa\ id_{()}$. Thus, we must show that $(return, \lambda f \to return\ (IdT \circ \pi \circ f)) \in (\mathcal{R} \to \kappa\ \mathcal{S})$. So for any $(f, f) \in \mathcal{R}$, we must show that $(return\ f, return\ (IdT \circ \pi \circ f)) \in \kappa\ \mathcal{S}$. As $(return, return) \in \mathcal{S} \to \kappa\ \mathcal{S}$, this amounts to showing that $(f, IdT \circ \pi \circ f) \in \mathcal{S}$. Take any $(b, b) \in id_{\beta,\beta}$, then $(f\ b, IdT \circ \pi\ \$\ f\ b) \in \mathcal{T}\ \kappa\ id_{()}$ should hold. In other words, $IdT \circ id \circ \pi\ \$\ f\ b \equiv IdT \circ \pi\ \$\ f\ b$ should hold. This is indeed true. Hence the assumption about (g, g') does hold. □

Here is the second auxiliary lemma.

Lemma 5 *Consider a function* $f :: \forall m.Monad\ m \Rightarrow \alpha \to m\ (\beta \to m\ \gamma)$*, then we have that:*

$$f_\kappa \equiv (out\ (return \circ out\ (return \circ unId) \circ unId))\ f_{Id}$$

Proof: Let $\mathcal{F} : m \Leftrightarrow Id$ be the $Monad$ action

$$\mathcal{F}\ \mathcal{R} = return^{-1}; \mathcal{R}; Id$$

. This is a $Monad$ action indeed, as was already shown by Voigtländer (see Voigtländer (2009), p.5, as part of the proof of Theorem 1). □

Here is the third auxiliary lemma.

Lemma 6 *Consider a function* $adv :: \forall m.Monad\ m \Rightarrow \alpha \to \tau\ m\ (\beta \to \tau\ m\ ())$*, then we have that:*

$$(out\ (IdT \circ fmap\ (out\ (IdT \circ \pi)) \circ \pi))\ adv \equiv const\ (return\ (const\ (return\ ())))$$

where τ *is a* $MonadTrans$*.*

Proof: The proof is depicted in Figure 13.

□

Define $\otimes$ as the counterpart of $\circledast$:

$$\begin{aligned}&(\otimes) :: \forall t\ m\ a\ b.(Monad\ m, MonadTrans\ t, Monad\ (t\ m))\\ &\quad \Rightarrow Augment\ a\ t\ m\ b\\ &\quad \to Open\ (a \to t\ m\ b)\\ &\quad \to Open\ (a \to t\ m\ b)\\ &advice \otimes base = around\ advice \perp base\end{aligned}$$

Here is the fifth auxiliary lemma.

Lemma 7 *Consider a function* $bse :: \forall t.MonadTrans\ t \Rightarrow Open\ (\alpha \to t\ m\ \beta)$*, then we have that:*

$$weave\ ((const\ (return\ (const\ (return\ ()))))) \otimes bse) \equiv weave\ bse$$

where κ *is a* $Monad$*.*

Proof: The proof is depicted in Figure 14.

□

B.1 Main Proof

The main theorem follows from the above lemmas.

Proof: The proof is depicted in Figure 15. Note that the proof relies on equational reasoning.

□

C. Proofs of Projection Functions

C.1 The $projW$ Function

Proof:

The proof is listed in Figure 16.

□

C.2 The $projS$ Function

Proof: The proof is listed in Figure 17.

□

D. Proof of Harmless Observation Advice

Again we turn to the same convenient intermediate form for augmentation advice that we used for the proof of orthogonal harmless advice. The define $\circledcirc$ as the counterpart of $\odot$ for:

$$\begin{aligned}&(\circledcirc) :: (MonadGet\ s\ m, MonadTrans\ t, Monad\ (t\ m))\\ &\quad \Rightarrow (a \to t\ m\ (b \to t\ m\ ()))\\ &\quad \to Open\ (a \to t\ m\ b)\\ &\quad \to Open\ (a \to t\ m\ b)\\ &advice \circledcirc base = around\ advice\ `observation`\ base\end{aligned}$$

Now we first formulate and prove a few lemmas before we proceed with the main proof.

Lemma 8 *Consider a function* $f :: \forall m.MonadGet\ \sigma\ m \Rightarrow \alpha \to m\ (\beta \to m\ \gamma)$*, then we have that:*

$$f_\kappa \equiv (out\ (\lambda m \to aux\ m \gg\!= return \circ out\ aux))\ f_{(Reader\ \sigma)}$$

where

$$\begin{aligned}&aux :: MonadGet\ s\ m \Rightarrow Reader\ s\ a \to m\ a\\ &aux\ m = get \gg\!= \lambda s \to return\ (runReader\ m\ s)\end{aligned}$$

Proof: Let $\mathcal{F} : \kappa \Leftrightarrow Reader\ \sigma$ be the $MonadGet\ \sigma$ action

$$\mathcal{F}\ \mathcal{R} = (get \gg\!=)^{-1}\ ;\ out\ return^{-1}\ ;\ id_\sigma \to \mathcal{R}\ ;\ Reader$$

.

This is indeed a $MonadGet\ \sigma$ action:

- Assume that $(a, b) \in \mathcal{R}$. We have that $(get \gg\!=)^{-1}\ (return_\kappa a) = const\ (return_\kappa a)$. Also, $out\ return^{-1}\ (const\ (return_\kappa a)) = const\ a$. Finally, note that $Reader\ (const\ b) = return_{Reader\ \sigma}\ b$. In conclusion, $(return_\kappa, return_{Reader\ \sigma}) \in \mathcal{R} \to \mathcal{F}\ \mathcal{R}$.
- For get_κ we do have that $(get \gg\!=)^{-1}\ get_\kappa = return_\kappa$, and $out\ return^{-1}\ return_\kappa = id$. Moreover, $id \circ id \circ id = id$. Finally, $Reader\ id = get_{Reader\ \sigma}$. Ergo, $(get_\kappa, get_{Reader\ \sigma}) \in \mathcal{F}\ id_\sigma$.
- For all $\mathcal{R}, \mathcal{S}, (f_1, f_2) \in id_\sigma \to \mathcal{R}$ and for all $(k_1, k_2) \in i\mathcal{R} \to id_\sigma \to \mathcal{S}$, We have that $(get \gg\!= \lambda s \to return\ (f_1\ s), get \gg\!= \lambda s \to return\ (f_2\ s)) \in \mathcal{F}\ \mathcal{R}$. Similarly, we have that $(\lambda x \to get \gg\!= \lambda s \to return\ (k_1\ x\ s), \lambda x \to get \gg\!= \lambda s \to return\ (k_2\ x\ s)) \in \mathcal{R} \to \mathcal{F}\ \mathcal{S}$. Moreover,

$$\begin{aligned}&get \gg\!= \lambda s \to return\ (f_1\ s)\\ \equiv\ &\{\text{-unfold } return\text{ -}\}\\ &get \gg\!= \lambda s \to Reader\ (const\ (f_1\ s))\\ \equiv\ &\{\text{-unfold } get\text{ -}\}\\ &Reader\ id \gg\!= \lambda s \to Reader\ (const\ (f_1\ s))\\ \equiv\ &\{\text{-unfold } \gg\!=\text{ -}\}\\ &Reader\ \$\ \lambda s \to\\ &\quad runReader\ (Reader\\ &\quad\quad (const\ (f_1\ (runReader\ (Reader\ id)\ s))))\ s\\ \equiv\ &\{\text{-}runReader\ (Reader\ f) \equiv f\text{ -}\}\\ &Reader\ \$\ \lambda s \to const\ (f_1\ (id\ s))\ s\\ \equiv\ &\{\text{-unfold } const\text{ -}\}\\ &Reader\ \$\ \lambda s \to f_1\ (id\ s)\\ \equiv\ &\{\text{-unfold } id\text{ -}\}\end{aligned}$$

$$
\begin{array}{ll}
 & (out\ (IdT \circ fmap\ (out\ (IdT \circ \pi)) \circ \pi))\ adv \\
\equiv & \{\text{-}out\ (f \circ g) \equiv out\ f \circ out\ g\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT \circ out\ \pi) \circ \pi))\ adv \\
\equiv & \{\text{-}fmap\ (g \circ h) \equiv fmap\ g \circ fmap\ h\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT) \circ fmap\ (out\ \pi) \circ \pi))\ adv \\
\equiv & \{\text{-}out\ (f \circ g) \equiv out\ f \circ out\ g\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)) \circ out\ (fmap\ (out\ \pi) \circ \pi))\ adv \\
\equiv & \{\text{-Unfolding of } (\circ)\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ (out\ (fmap\ (out\ \pi) \circ \pi)\ adv) \\
\equiv & \{\text{-Lemma 5 -}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (return \circ out\ (return \circ unId) \circ unId))\ (out\ (fmap\ (out\ \pi) \circ \pi)\ adv)) \\
\equiv & \{\text{-}out\ (f \circ g) \equiv out\ f \circ out\ g\ (\times 3)\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (return \circ out\ return)\ (out\ (out\ unId \circ unId)\ (out\ (fmap\ (out\ \pi) \circ \pi)\ adv)))) \\
\equiv & \{\text{-Totality assumption -}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (return \circ out\ return))\ (const\ (const\ ()))) \\
\equiv & \{\text{-Unfolding of } (\circ) \text{ and } out\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ (const\ (return\ (const\ (return\ ())))) \\
\equiv & \{\text{-Unfolding of } out\ \text{-}\} \\
 & const\ ((IdT \circ fmap\ (out\ IdT) \circ return)\ (const\ (return\ ()))) \\
\equiv & \{\text{-}fmap\ h \circ return \equiv return \circ h\ \text{-}\} \\
 & const\ ((IdT \circ return \circ out\ IdT)\ (const\ (return\ ()))) \\
\equiv & \{\text{-}IdT \circ return \equiv return\ \text{-}\} \\
 & const\ ((return \circ out\ IdT)\ (const\ (return\ ()))) \\
\equiv & \{\text{-}out\ f\ (const\ x) \equiv const\ (f\ x)\ \text{-}\} \\
 & const\ (return\ (const\ (IdT\ (return\ ())))) \\
\equiv & \{\text{-}IdT \circ return \equiv return\ \text{-}\} \\
 & const\ (return\ (const\ (return\ ())))
\end{array}
$$

Figure 13. Proof of Lemma 6

$$
\begin{array}{ll}
 & weave\ ((const\ (return\ (const\ (return\ ())))) \otimes bse) \\
\equiv & \{\text{-unfold def. of } \otimes\ \text{-}\} \\
 & weave\ (\lambda p\ x \rightarrow const\ (return\ (const\ (return\ ())))\ x \gg\!= \lambda aft \rightarrow bse\ p\ x \gg\!= \lambda r \rightarrow aft\ r \gg\!= \backslash_ \rightarrow return\ r) \\
\equiv & \{\text{-unfold } const\ \text{-}\} \\
 & weave\ (\lambda p\ x \rightarrow return\ (const\ (return\ ()))\ \gg\!= \lambda aft \rightarrow bse\ p\ x \gg\!= \lambda r \rightarrow aft\ r \gg\!= \backslash_ \rightarrow return\ r) \\
\equiv & \{\text{-Monad left unit -}\} \\
 & weave\ (\lambda p\ x \rightarrow bse\ p\ x \gg\!= \lambda r \rightarrow const\ (return\ ())\ r \gg\!= \backslash_ \rightarrow return\ r) \\
\equiv & \{\text{-unfold } const\ \text{-}\} \\
 & weave\ (\lambda p\ x \rightarrow bse\ p\ x \gg\!= \lambda r \rightarrow return\ ()\ \gg\!= \backslash_ \rightarrow return\ r) \\
\equiv & \{\text{-Monad left unit -}\} \\
 & weave\ (\lambda p\ x \rightarrow bse\ p\ x \gg\!= \lambda r \rightarrow return\ r) \\
\equiv & \{\text{-eta reduction -}\} \\
 & weave\ (\lambda p\ x \rightarrow bse\ p\ x \gg\!= return) \\
\equiv & \{\text{-Monad right unit -}\} \\
 & weave\ (\lambda p\ x \rightarrow bse\ p\ x) \\
\equiv & \{\text{-eta reduction -}\} \\
 & weave\ bse
\end{array}
$$

Figure 14. Proof of Lemma 7

$\pi \circ weave\ ((bef, aft) \circledast bse)$
$\equiv$ { Lemma 3 }
$\pi \circ weave\ (convert\ (bef, aft) \otimes bse)$
$\equiv$ { **let** $adv = convert\ (bef, aft)$ }
$\pi \circ weave\ (adv \otimes bse)$
$\equiv$ { inverse beta reduction }
$\pi \circ ((\lambda x \to weave\ (x \otimes bse))\ adv)$
$\equiv$ { fold out }
$(out\ \pi \circ (\lambda x \to weave\ (x \otimes bse)))\ adv$
$\equiv$ { Lemma 4 }
$(out\ unIdT \circ (\lambda x \to weave\ (x \otimes bse)) \circ out\ (IdT \circ fmap\ (out\ (IdT \circ \pi)) \circ \pi))\ adv$
$\equiv$ { Unfolding of $(\circ)$ }
$(out\ unIdT \circ (\lambda x \to weave\ (x \otimes bse)))\ ((out\ (IdT \circ fmap\ (out\ (IdT \circ \pi)) \circ \pi))\ adv)$
$\equiv$ { Lemma 6 }
$(out\ unIdT \circ (\lambda x \to weave\ (x \otimes bse)))\ (const\ (return\ (const\ (return\ ()))))$
$\equiv$ { unfold def. of $(\circ)$ }
$out\ unIdT\ ((\lambda x \to weave\ (x \otimes bse))\ (const\ (return\ (const\ (return\ ()))))$
$\equiv$ { beta reduction }
$out\ unIdT\ (weave\ ((const\ (return\ (const\ (return\ ())))) \otimes bse))$
$\equiv$ { Lemma 7 }
$out\ unIdT\ (weave\ bse)$

Figure 15. Proof of the Harmless Advice Theorem

$projW \circ lift$
$\equiv$ {-unfold $\circ$ -}
$(\lambda m \to projW\ (lift\ m))$
$\equiv$ {-unfold $lift$ -}
$(\lambda m \to projW\ (WriterT\ (m \gg\!= \lambda x \to return\ (x, mempty))))$
$\equiv$ {-unfold $projW$ -}
$(\lambda m \to runWriterT\ (WriterT\ (m \gg\!= \lambda x \to return\ (x, mempty))) \gg\!= return \circ fst)$
$\equiv$ {-$runWriterT\ (WriterT\ m) \equiv m$ -}
$(\lambda m \to m \gg\!= \lambda x \to return\ (x, mempty) \gg\!= return \circ fst)$
$\equiv$ {-Monad left unit -}
$(\lambda m \to m \gg\!= \lambda x \to (return \circ fst)\ (x, mempty))$
$\equiv$ {-unfold $\circ$ -}
$(\lambda m \to m \gg\!= \lambda x \to return\ (fst\ (x, mempty)))$
$\equiv$ {-unfold fst -}
$(\lambda m \to m \gg\!= \lambda x \to return\ x)$
$\equiv$ {-eta reduction -}
$(\lambda m \to m \gg\!= return)$
$\equiv$ {-Monad right unit -}
$(\lambda m \to m)$
$\equiv$ {-fold id -}
id

Figure 16. Proof of $projW \circ lift \equiv id$.

$$
\begin{array}{ll}
& projS\ s0 \circ lift \\
\equiv & \{\text{-unfold} \circ \text{-}\} \\
& (\lambda m \to projS\ s0\ (lift\ m)) \\
\equiv & \{\text{-unfold } lift\ \text{-}\} \\
& (\lambda m \to projS\ s0\ (StateT\ (\lambda s \to m \gg\!= \lambda x \to return\ (x, s)))) \\
\equiv & \{\text{-unfold } projS\ \text{-}\} \\
& (\lambda m \to runStateT\ (StateT\ (\lambda s \to m \gg\!= \lambda x \to return\ (x, s)))\ s0 \gg\!= return \circ fst) \\
\equiv & \{\text{-}runStateT\ (StateT\ f) \equiv f\ \text{-}\} \\
& (\lambda m \to (\lambda s \to m \gg\!= \lambda x \to return\ (x, s))\ s0 \gg\!= return \circ fst) \\
\equiv & \{\text{-beta reduction -}\} \\
& (\lambda m \to m \gg\!= \lambda x \to return\ (x, s0) \gg\!= return \circ fst) \\
\equiv & \{\text{-Monad left unit -}\} \\
& (\lambda m \to m \gg\!= \lambda x \to (return \circ fst)\ (x, s0)) \\
\equiv & \{\text{-unfold} \circ \text{-}\} \\
& (\lambda m \to m \gg\!= \lambda x \to return\ (fst\ (x, s0))) \\
\equiv & \{\text{-unfold } fst\ \text{-}\} \\
& (\lambda m \to m \gg\!= \lambda x \to return\ x) \\
\equiv & \{\text{-eta reduction -}\} \\
& (\lambda m \to m \gg\!= return) \\
\equiv & \{\text{-Monad right unit -}\} \\
& (\lambda m \to m) \\
\equiv & \{\text{-fold } id\ \text{-}\} \\
& id
\end{array}
$$

Figure 17. Proof of $projS \circ lift \equiv id$.

$$
\begin{array}{ll}
& Reader\ \$\ \lambda s \to f_1\ s \\
\equiv & \{\text{-eta reduction -}\} \\
& Reader\ f_1
\end{array}
$$

Similarly, we can show that

$$
\begin{array}{ll}
& \lambda x \to get \gg\!= \lambda s \to return\ (k_2\ x\ s) \\
\equiv & \{\text{-... -}\} \\
& \lambda x \to Reader\ (k_2\ x)
\end{array}
$$

Now consider $(get \gg\!= \lambda s \to return\ (f_1\ s) \gg\!= \lambda x \to get \gg\!= \lambda s' \to return\ (k_1\ x\ s'), Reader\ f_2 \gg\!= \lambda x \to Reader\ (k_2\ x))$. We can rewrite the first component

$$
\begin{array}{ll}
& get \gg\!= \lambda s \to return\ (f_1\ s) \gg\!= \lambda x \to \\
& get \gg\!= \lambda s' \to return\ (k_1\ x\ s') \\
\equiv & \{\text{-Monad left unit -}\} \\
& get \gg\!= \lambda s \to get \gg\!= \lambda s' \to return\ (k_1\ (f_1\ s)\ s') \\
\equiv & \{\text{-get idempotence -}\} \\
& get \gg\!= \lambda s \to return\ (k_1\ (f_1\ s)\ s)
\end{array}
$$

If we apply $(get \gg\!=)^{-1}\ ;\ out\ \ return^{-1}$ to this, we get $\lambda s \to (k_1\ (f_1\ s)\ s)$.

Similarly, we can rewrite the second component

$$
\begin{array}{ll}
& Reader\ f_2 \gg\!= \lambda x \to Reader\ (k_2\ x) \\
\equiv & \{\text{-unfold} \gg\!= \text{-}\} \\
& Reader\ \$\ \lambda s \to runReader \\
& \quad (Reader\ (k_2\ (runReader\ (Reader\ f_2)\ s)))\ s \\
\equiv & \{\text{-}runReader\ (Reader\ f) \equiv f\ \text{-}\} \\
& Reader\ \$\ \lambda s \to runReader\ (Reader\ (k_2\ (f_2\ s)))\ s \\
\equiv & \{\text{-}runReader\ (Reader\ f) \equiv f\ \text{-}\} \\
& Reader\ \$\ \lambda s \to k_2\ (f_2\ s)\ s
\end{array}
$$

Summarizing, the original pair is in $\mathcal{F}\ \mathcal{S}$. Hence, we have that $(\gg\!=_\kappa, \gg\!=_{Reader\ \sigma}) \in \mathcal{F}\ \mathcal{R} \to (\mathcal{R} \to \mathcal{F}\ \mathcal{S}) \to \mathcal{F}\ \mathcal{S}$.

Note that the function aux captures $\mathcal{F}\ id$. The theorem follows.

□

The next lemma is the counterpart of Lemma 6.

Lemma 9 *Consider a function* $adv :: \forall m.MonadGet\ \sigma\ m \Rightarrow \alpha \to \tau\ m\ (\beta \to \tau\ m\ ())$, *then we have that:*

$$
\begin{array}{c}
(out\ (IdT \circ fmap\ (out\ (IdT \circ \pi)) \circ \pi))\ adv \\
\equiv \\
const\ (return\ (const\ (return\ ())))
\end{array}
$$

where τ *is a* $MonadTrans$.

Proof: The proof is depicted in Figure 18.

□

Lemma 10 *Consider a function* $bse :: \forall t.MonadTrans\ t \Rightarrow Open\ (\alpha \to t\ m\ \beta)$, *then we have that:*

$$
\begin{array}{c}
weave\ ((const\ (return\ (const\ (return\ ())))) \circledcirc bse) \\
\equiv \\
weave\ bse
\end{array}
$$

where κ *is a* $Monad$.

Proof: The proof is depicted in Figure 19.

□

The main proof is similar to the Harmless Advice proof. The only difference lies in the use of Lemma 9, which relies on the $MonadGet$ law.

Proof: □

$$
\begin{array}{ll}
 & (out\ (IdT \circ fmap\ (out\ (IdT \circ \pi)) \circ \pi))\ adv \\
\equiv & \{\text{-}out\ (f \circ g) \equiv out\ f \circ out\ g\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT \circ out\ \pi) \circ \pi))\ adv \\
\equiv & \{\text{-}fmap\ (g \circ h) \equiv fmap\ g \circ fmap\ h\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT) \circ fmap\ (out\ \pi) \circ \pi))\ adv \\
\equiv & \{\text{-}out\ (f \circ g) \equiv out\ f \circ out\ g\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)) \circ out\ (fmap\ (out\ \pi) \circ \pi))\ adv \\
\equiv & \{\text{-Unfolding of } (\circ)\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ (out\ (fmap\ (out\ \pi) \circ \pi)\ adv) \\
\equiv & \{\text{-}\mathbf{let}\ adv' = (out\ (fmap\ (out\ \pi) \circ \pi)\ adv)\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ adv' \\
\equiv & \{\text{-Lemma 8 -}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (\lambda m \rightarrow aux\ m \gg\!= return \circ out\ aux))\ adv') \\
\equiv & \{\text{-unfold } aux\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (\lambda m \rightarrow aux\ m \gg\!= return \circ out\ (\lambda n \rightarrow get \gg\!= \lambda s \rightarrow return\ (runReader\ n\ s))))\ adv') \\
\equiv & \{\text{-Totality assumption -}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (\lambda m \rightarrow aux\ m \gg\!= return \circ out\ (\lambda n \rightarrow get \gg\!= \lambda s \rightarrow return\ ()))\ adv') \\
\equiv & \{\text{-}MonadGet\ \text{law -}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (\lambda m \rightarrow aux\ m \gg\!= return \circ out\ (\lambda n \rightarrow return\ ()))\ adv') \\
\equiv & \{\text{-fold } const\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (\lambda m \rightarrow aux\ m \gg\!= return \circ out\ (const\ (return\ ())))\ adv') \\
\equiv & \{\text{-unfold } aux\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (\lambda m \rightarrow get \gg\!= \lambda s \rightarrow return\ (runReader\ m\ s) \gg\!= return \circ out\ (const\ (return\ ())))\ adv') \\
\equiv & \{\text{-Monad left unit -}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (\lambda m \rightarrow get \gg\!= \lambda s \rightarrow return\ (out\ (const\ (return\ ()))\ (runReader\ m\ s))))\ adv') \\
\equiv & \{\text{-}out\ (const\ x)\ y \equiv const\ y\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (\lambda m \rightarrow get \gg\!= \lambda s \rightarrow return\ (const\ (return\ ())))\ adv') \\
\equiv & \{\text{-}MonadGet\ \text{law -}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (\lambda m \rightarrow return\ (const\ (return\ ())))\ adv') \\
\equiv & \{\text{-fold } const\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ ((out\ (const\ (return\ (const\ (return\ ()))))\ adv') \\
\equiv & \{\text{-}out\ (const\ x)\ y \equiv const\ y\ \text{-}\} \\
 & (out\ (IdT \circ fmap\ (out\ IdT)))\ (const\ (return\ (const\ (return\ ())))) \\
\equiv & \{\text{-}out\ (f \circ g) \equiv out\ f \circ out\ g\ \text{-}\} \\
 & (out\ IdT \circ out\ (fmap\ (out\ IdT)))\ (const\ (return\ (const\ (return\ ())))) \\
\equiv & \{\text{-unfold} \circ \text{-}\} \\
 & out\ IdT\ (out\ (fmap\ (out\ IdT))\ (const\ (return\ (const\ (return\ ()))))) \\
\equiv & \{\text{-out f (const x) == const (f x) -}\} \\
 & out\ IdT\ (const\ (fmap\ (out\ IdT)\ (return\ (const\ (return\ ()))))) \\
\equiv & \{\text{-}fmap\ f\ (return\ x) = return\ (f\ x)\ \text{-}\} \\
 & out\ IdT\ (const\ (return\ (out\ IdT\ (const\ (return\ ()))))) \\
\equiv & \{\text{-}out\ f\ (const\ x) \equiv const\ (f\ x)\ \text{-}\} \\
 & out\ IdT\ (const\ (return\ (const\ (IdT\ (return\ ()))))) \\
\equiv & \{\text{-}IdT\ (return\ x) \equiv return\ x\ \text{-}\} \\
 & out\ IdT\ (const\ (return\ (const\ (return\ ())))) \\
\equiv & \{\text{-out f (const x) == const (f x) -}\} \\
 & const\ (IdT\ (return\ (const\ (return\ ())))) \\
\equiv & \{\text{-}IdT\ (return\ x) \equiv return\ x\ \text{-}\} \\
 & const\ (return\ (const\ (return\ ())))
\end{array}
$$

Figure 18. Proof of Lemma 9

$$
\begin{array}{ll}
 & \mathit{weave}\ ((\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))) \circledcirc \mathit{bse}) \\
\equiv & \{\text{-unfold def. of } \circledcirc \text{ -}\} \\
 & \mathit{weave}\ (\lambda p\ x \to \mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))\ x \gg\!= \lambda \mathit{aft} \to \mathit{bse}\ p\ x \gg\!= \lambda r \to \mathit{aft}\ r \gg\!= \backslash_ \to \mathit{return}\ r) \\
\equiv & \{\text{-fold def. of } \otimes \text{ -}\} \\
 & \mathit{weave}\ ((\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))) \otimes \mathit{bse}) \\
\equiv & \{\text{-Lemma 7 -}\} \\
 & \mathit{weave}\ \mathit{bse}
\end{array}
$$

Figure 19. Proof of Lemma 10

$$
\begin{array}{ll}
 & \pi \circ \mathit{weave}\ ((\mathit{bef}, \mathit{aft}) \odot \mathit{bse}) \\
\equiv & \{\text{-Lemma 3 -}\} \\
 & \pi \circ \mathit{weave}\ (\mathit{convert}\ (\mathit{bef}, \mathit{aft}) \circledcirc \mathit{bse}) \\
\equiv & \{\text{-}\mathbf{let}\ \mathit{adv} = \mathit{convert}\ (\mathit{bef}, \mathit{aft})\ \text{-}\} \\
 & \pi \circ \mathit{weave}\ (\mathit{adv} \circledcirc \mathit{bse}) \\
\equiv & \{\text{-inverse beta reduction -}\} \\
 & \pi \circ ((\lambda x \to \mathit{weave}\ (x \circledcirc \mathit{bse}))\ \mathit{adv}) \\
\equiv & \{\text{-fold } \mathit{out}\ \text{-}\} \\
 & (\mathit{out}\ \pi \circ (\lambda x \to \mathit{weave}\ (x \circledcirc \mathit{bse})))\ \mathit{adv} \\
\equiv & \{\text{-Lemma 4 -}\} \\
 & (\mathit{out}\ \mathit{unIdT} \circ (\lambda x \to \mathit{weave}\ (x \circledcirc \mathit{bse})) \circ \mathit{out}\ (\mathit{IdT} \circ \mathit{fmap}\ (\mathit{out}\ (\mathit{IdT} \circ \pi)) \circ \pi))\ \mathit{adv} \\
\equiv & \{\text{-Unfolding of } (\circ) \text{ -}\} \\
 & (\mathit{out}\ \mathit{unIdT} \circ (\lambda x \to \mathit{weave}\ (x \circledcirc \mathit{bse})))\ ((\mathit{out}\ (\mathit{IdT} \circ \mathit{fmap}\ (\mathit{out}\ (\mathit{IdT} \circ \pi)) \circ \pi))\ \mathit{adv}) \\
\equiv & \{\text{-Lemma 9 -}\} \\
 & (\mathit{out}\ \mathit{unIdT} \circ (\lambda x \to \mathit{weave}\ (x \circledcirc \mathit{bse})))\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))) \\
\equiv & \{\text{-unfold def. of } (\circ) \text{ -}\} \\
 & \mathit{out}\ \mathit{unIdT}\ ((\lambda x \to \mathit{weave}\ (x \circledcirc \mathit{bse}))\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))) \\
\equiv & \{\text{-beta reduction -}\} \\
 & \mathit{out}\ \mathit{unIdT}\ (\mathit{weave}\ ((\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))) \circledcirc \mathit{bse})) \\
\equiv & \{\text{-Lemma 10 -}\} \\
 & \mathit{out}\ \mathit{unIdT}\ (\mathit{weave}\ \mathit{bse})
\end{array}
$$

Figure 20. Proof of the Harmless Observation Advice Theorem