EffectiveAdvice: Disciplined Advice with Explicit Effects

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. 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 each other’s control and data flows.

This paper presents EffectiveAdvice, a disciplined model of advice, inspired by Aldrich’s Open Modules, that has full support for effects. With EffectiveAdvice, equivalence of advice, as well as base components, can be checked by equational reasoning. The paper describes EffectiveAdvice as a Haskell library in which 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 semantic model of advice that supports effects, and allows these effects to be separated with strong non-interference guarantees, or merged as needed.

Categories and Subject Descriptors

D.3.2 [Programming Languages]: Language Classifications—Functional Languages; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs

General Terms

Languages

Keywords

Mixins, monads, AOP, parametricity, interference

1. INTRODUCTION

In many specialized forms of modularity, including aspect-oriented programming (AOP) [17], feature-oriented programming (FOP) [30], and object-oriented programming (OOP) inheritance [7], 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 [33, 16, 1]. The latter problem, usually referred to as interference, has also received much attention in the AOP literature [32, 11, 8, 6]. 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.

Kiczales and Mezini [16] argue that modular reasoning about AOP, and similar mechanisms that capture crosscutting concerns is hard, and that a degree of global analysis may always be needed. In contrast, Aldrich [1] presents a pure functional core 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.

This paper presents EffectiveAdvice, a semantic model of advice that is inspired by Aldrich’s Open Modules and retains similar reasoning properties. However, unlike Open Modules, effects are fully supported through the use of monads [37]. Our model has close similarities with the monadic model of FOP proposed by Prehofer [51]. However an impor-
tant difference is the use of open recursion to model tightly coupled components, which are not considered in the basic model by Prehofer. Like Ligatti et al. [21], and even Aldrich [1] to some extent, we propose a non-oblivious core language with explicit advice points and explicit advice composition. Consequently, our core language cannot be viewed as an AOP language in the traditional sense [13], although it can be viewed, more generally, as a different approach to modularizing crosscutting concerns. Nonetheless, as we shall see, languages like Scala provide linguistic support for our model of advice and enjoy limited obliviousness.

EffectiveAdvice promotes the idea that effects should be an integral part of the interfaces of components, and that no implicit effects should occur. The programming model is based on open recursion, explicit advice points, and a requirement for every component to state the effects that it uses. Unlike Ligatti et al. [21], we do not devise a novel core language, but reuse the well-studied polymorphic λ-calculus, System F, extended with recursion and benefit from the many established technical results. Like other authors [36, 35], we use Haskell as a convenient source language for System F and elaborate EffectiveAdvice as a Haskell library\(^1\). Mixin composition is used to weave advice into a base program [7]. Monads [37] model effects and, for compositionality, non-monadic functions must be lifted into a monad.

In the purely functional model for EffectiveAdvice, equivalence of advice, as well as base programs, is determined by equational reasoning. Different interference patterns [32] between advice and base programs, constraining possible data and control flow interactions, can be enforced through the use of combinators. Higher-rank types [28] are used to ensure non-interference of effects. A key novelty introduced by EffectiveAdvice is the use of parametricity [36, 35], a powerful modular reasoning technique based on types only, to prove theorems for combinators providing strong guarantees of non-interference. Parametricity is used to prove two harmless advice [8] theorems, allowing a precise formulation of non-interference results without looking at the implementation of programs and which can be easily extended to cover new kinds of effects.

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 non-interference guarantees for control and data flow through the use of combinators and the type system. This is in contrast to other approaches, which usually achieve similar results by imposing syntactic constraints [31, 5, 29] or using type and effect systems that ensure such non-interference properties [8, 6].

- A novel use of parametricity to reason about non-interference of effects between components, which is used to prove theorems for harmless advice and harmless observation advice. In Section 7 a detailed compari-

---

\(^1\)www.cs.kuleuven.be/~toms/EffectiveAdvice.tgz

Figure 1: Basic mixin combinators.

The proofs of the theorems and background information are available in a technical report [26].

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 [7]. The connection between mixins and aspects is known [22]. 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 Open s is a synonym for a function with type s → 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 ∪ defines component (or advice) composition. Composition is associative, and it has the zero component as left and right units of ∪, forming a monoid. Note that this is just the monoid of endofunctions with identity and function composition.

\[ f \cup zero \equiv f \land zero \cup f \]
\[ (f \cup g) \cup h \equiv f \cup (g \cup h) \]

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

Consider the following open functions:

\[ fb_1 :: \text{Open} (\text{Int} \rightarrow \text{Int}) \]
\[ fb_1 \text{ proceed } n = \text{case } n \text{ of} \]
\[ 0 \rightarrow 0 \]
\[ 1 \rightarrow 1 \]
\[ \rightarrow \text{proceed } (n - 1) + \text{proceed } (n - 2) \]

\[ adefb :: \text{Open} (\text{Int} \rightarrow \text{Int}) \]
\[ adefb \text{ proceed } n = \text{case } n \text{ of} \]
\[ 0 \rightarrow 55 \]
\[ 30 \rightarrow 832040 \]
\[ \rightarrow \text{proceed } n \]
The open function $fib_1$ defines the standard Fibonacci function, except that recursive calls are replaced by $\texttt{proceed}$. The open function $\texttt{ade}fib$ optimizes two calls of the Fibonacci function by returning the appropriate values immediately. Note that $\texttt{ade}fib$ 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:

$$\texttt{slowfib}_{\texttt{optfib}} : \texttt{Int} \to \texttt{Int}$$
$$\texttt{slowfib}_{\texttt{optfib}} = \texttt{weave} (\texttt{fib}_1)$$

The functions $\texttt{slowfib}$ and $\texttt{optfib}$ 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 $\texttt{proceed}$ is understood as a recursive call, while in advice $\texttt{proceed}$ 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 $\texttt{proceed}$ calls in a chain of advice being composed.

$$\texttt{proceed}$$

$$p = \texttt{weave} (a_1 \oplus a_2 \oplus \ldots \oplus a_n \oplus \texttt{base})$$

In the advice $a_1$ the $\texttt{proceed}$ reference is pointing to $a_2$ (the next advice in the chain); in the $a_2$ advice proceed points to the next advice in the chain and so on for the other advice. When the base program is reached, $\texttt{proceed}$ just points back to the beginning of the advice chain. The behavior of $\texttt{proceed}$ for advice and base programs are, respectively, akin to $\texttt{super}$ and $\texttt{this}$ in an OO language.

In the Haskell approach presented in this section, the $\texttt{proceed}$ argument to advice or base programs is always explicitly passed. However, it is possible to make $\texttt{proceed}$ implicit using implicit parameters [19].

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 $\texttt{Open}$. 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.

## 2.2 Monads as Explicit Effects

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. Effective advice is helpful to provide a better solution for this problem, allowing the creation of a dynamic memo table where previously computed calls can be looked up.

EffectiveAdvice models effects using monads and monad transformers. For reasons of space, an introduction to these concepts is not presented in this paper. However, one such introduction can be found in the technical report [26] and there are several other, more complete, resources available [37, 38, 20]. We summarize the essential monad definitions used throughout the paper in Figure 2.

A simple effectful memoization advice is presented in Figure 3. The $\texttt{MemoState}$ class, which models state, is used by the $\texttt{memo}$ aspect to read and update the cached values in the memo table. The memo table is implemented using a map from integers to integers.\footnote{empty denotes an empty map, insert inserts a key-value pair, and (!) looks up the value for a given key.} 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 2, to recover variations of the
for any client? Obviously it is not possible to show this for the fibonacci function. For example, the identity monad recovers the effect-free function
\[ \text{slowfib}_2 :: \text{Int} \rightarrow \text{Int} \]
\[ \text{slowfib}_2 = \text{runUnit} \circ \text{weave} \circ \text{fib}_2 \]
while a fast fibonacci function is obtained by adding the memo advice and suitably instantiating the state monad:
\[ \text{evalState} :: \text{State} \text{a} \rightarrow s \rightarrow \text{a} \]
\[ \text{evalState} (\text{State} f) s = \text{fst} (f s) \]
\[ \text{fastfib} :: \text{Int} \rightarrow \text{Int} \]
\[ \text{fastfib} n = \text{evalState} (\text{weave} (\text{memo} \oplus \text{fib}_2) n) \text{empty} \]

**Equational Reasoning** 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 \( \text{fib}_2 \):

\[ \text{fib}_3 \text{ proceed } n = \text{case } n \text{ of} \]
\[ 0 \rightarrow \text{return } 0 \]
\[ 1 \rightarrow \text{return } 1 \]
\[ \rightarrow \text{do } y \leftarrow \text{proceed } (n - 2) \]
\[ x \leftarrow \text{proceed } (n - 1) \]
\[ \text{return } (x + y) \]

\[ \text{fib}_4 \text{ proceed } n = \text{case } n \text{ of} \]
\[ 0 \rightarrow \text{return } 0 \]
\[ 1 \rightarrow \text{return } 1 \]
\[ \rightarrow \text{do } m \leftarrow \text{return } (n - 1) \]
\[ x \leftarrow \text{proceed } m \]
\[ y \leftarrow \text{proceed } (n - 1) \]
\[ \text{return } (x + y) \]

Are these two variants indistinguishable with respect to \( \text{fib}_2 \)? Obviously it is not possible to show this for \( \text{fib}_3 \), 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 \( \text{fib}_2 \equiv \text{fib}_4 \):

\[
\begin{align*}
\text{do } m &\leftarrow \text{return } (n - 1) \\
x &\leftarrow \text{proceed } m \\
y &\leftarrow \text{proceed } (n - 1) \\
\text{return } (x + y) \\
\equiv & \{ \text{Monad left unit: } \text{return } x \Rightarrow f \ x \ . \} \\
\text{do } x &\leftarrow \text{proceed } (n - 1) \\
y &\leftarrow \text{proceed } (n - 1) \\
\text{return } (x + y) \\
\equiv & \{ n - 1 + n - 2 \} \\
\text{do } x &\leftarrow \text{proceed } (n - 1) \\
y &\leftarrow \text{proceed } (n - 2) \\
\text{return } (x + y) \\
\end{align*}
\]

The same approach shows that the two pure functions \( \text{slowfib}_2 \) and \( \text{fastfib} \), 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 \( \text{EvenOdd} \) is the signature for a pair of two functions \( \text{even} \) and \( \text{odd} \) and the component \( \text{evenodd} \) provides an implementation. The advice \( \text{logEO} \) adds logging to those definitions (the implementation of \( \text{log} \) is shown in Figure 7). Logged versions of \( \text{even} \) and \( \text{odd} \) are recovered through weaving as follows:

\[
\begin{align*}
\text{leven} &\equiv \text{runWriter} \circ \text{even} \circ \text{weave} (\text{logEO} \oplus \text{evenodd}) \\
\text{loddd} &\equiv \text{runWriter} \circ \text{odd} \circ \text{weave} (\text{logEO} \oplus \text{evenodd}) \\
\end{align*}
\]

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 [16] noted this gap and concluded that the restrictions in terms of expressiveness in such an approach may just be too limiting. Indeed, the lack of effects is extremely limiting for practical applications. In this section, we show how EffectiveAdvice scales to a much more realistic setting that: 1) allows effects on both base programs and advice, and 2) handles multiple kinds of effects, such as state or exceptions.

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 \( \text{Open} (\text{Expr} \rightarrow m \text{Int}) \) can be understood in the by now familiar way: it exports a function of type \( \text{Expr} \rightarrow m \text{Int} \) 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 \( \text{Monad} m \) for an unknown type of effect. Instead it is \( \text{MonadState} n m \): the effect must involve an updateable state of type \( n \) in the environment used by the interpreter. In other words, the interpreter itself is effectful. In dealing with the \( \text{Var} \) and
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 e of
  Just x → return x
  _ → error msg
  Plus l r → do x ← proceed l
ey ← 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.

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

Figure 7: The logging aspect.

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, due to its role in imperative languages, is the most widely used and well-known type of effect. However, there are many other useful types of effect. EffectiveAdvice allows any effect expressible as a monad, including output streams, exceptions, I/O, non-determinism, and combinations 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 2 for writing logging messages.

Dumpping 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 ordures the case for variables and replaces the error primitive by throwException (see Figure 2). There are two advantages of using throwException 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:

dump1, dump2 :: (MonadWriter String m, MonadState Env m) ⇒ Expr → m Int
dump1 = weave (log *eval* ⊕ beval)
dump2 = weave (log *eval* ⊕ dump ⊕ beval)
exc :: (MonadError Exc m, MonadWriter String m, MonadState Env m) ⇒ Expr → m Int
exc = weave (eval ⊕ log *eval* ⊕ beval)

The dump1 program adds logging of function calls to the evaluator, while dump2 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

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

Figure 9: The exception handling aspect.
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 ++
  "\nExpression: " ++ 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. [32] 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 \( \text{Open} \ s \).
Replacement The informal requirement for replacement is that no calls are made to proceed. This requirement can be captured by the following combinator:

\[
\text{type Replace} \quad s = s
\]
\[
\text{replace} :: \text{Replace} \ s \to \text{Open} \ s
\]
\[
\text{replace rep} = \lambda \text{proceed} \to \text{radv}
\]

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

\[
\text{type Augment} \quad a \ b \ c \ m = (a \to m \ c, a \to b \to c \to m ()
\]
\[
\text{augment} :: \text{Monad} \ m
\]
\[
\Rightarrow \text{Augment} \ a \ b \ c \ m \Rightarrow \text{Open} \ (a \to m \ b)
\]
\[
\text{augment} \ (\text{bef}, \text{aft}) \text{proceed} \ a = \text{do} \{ \text{c Xin Expression: } e \}
\]
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 afterwords. 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, \( \log \), is logging advice

\[
\text{log} :: (\text{MonadWriter} \ String, \text{Show} \ a, \text{Show} \ b)
\]
\[
\Rightarrow \text{String} \to \text{Augment} \ a \ b ()
\]
\[
\text{log}, \text{name} = (\text{bef}, \text{aft}) \text{where}
\]
\[
\text{bef} \ x = \text{write} \ "\text{Entering} \ " \ x
\]
\[
\text{aft} \ x = \text{write} \ "\text{Exiting} \ " \ y
\]
\[
\text{write} \ a \ b = \text{tell} \ (a \oplus \text{name} \oplus \text{show} \ b \oplus "\text{\n"})
\]

Such that \( \text{log} \equiv \text{augment} \circ \text{log} \).

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

\[
\text{before} :: \text{Monad} \ m \Rightarrow (a \to m () \to \text{Open} \ (a \to m \ b)
\]
\[
\text{after} :: \text{Monad} \ m \Rightarrow (a \to \text{Open} \ (a \to m \ b)
\]
\[
\text{before} \ \text{b Xin Expression: } e
\]
\[
\text{after} \ \text{aft} \ \text{proceed} \ (\text{aft} \ (\text{bef} \ a \to \text{return} () \text{,a} \ b \ c \to \text{return} ())
\]

Our earlier dumping advice can be written as before advice:

\[
\text{dump1} :: (\text{MonadState} \ s \ m, \text{MonadWriter} \ String \ m, \text{Show} \ s)
\]
\[
\Rightarrow a \to m ()
\]
\[
\text{dump1} \text{ arg} = \text{do} \ s \to \text{get}
\]
\[
\text{tell} \ (\text{show} \ s \oplus "\text{\n"})
\]

Note that \( \text{dump} \equiv \text{before dump1} \).

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

\[
\text{type Narrow} \ a \ b \ c \ m =
\]
\[
(a \to m \ \text{Bool} \text{,Augment} \ a \ b \ c \ m, \text{Replace} \ (a \to m \ b))
\]
\[
\text{narrow} :: \text{Monad} \ m \Rightarrow \text{Narrow} \ a \ b \ c \ m \Rightarrow \text{Open} \ (a \to m \ b)
\]
\[
\text{narrow} \ (\text{p}, \text{aug}, \text{rep}) \text{proceed} \ x =
\]
\[
\text{if} \ b \ \text{then augment aug proceed x}
\]
\[
\text{else replace rep proceed x}
\]

The runtime choice is made by the predicate of type \( a \to m \ \text{Bool} \), 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.

\[
\text{memor} \equiv (\text{Monad} \, \text{State} \, (\text{Map} \, a \, b) \, m, \text{Ord} \, a)
\]

\[
\Rightarrow \text{Monad} \, a \, b \, (m) \, m
\]

\[
\text{memor} \equiv (p, (\text{bef}, \text{aft}), \text{rep}) \text{ where}
\]

\[
p \, x \quad = \quad \text{do} \{(m \leftarrow \text{get}; \text{return} \, (\text{member} \, x \, m))\}
\]

\[
\text{bef} \, x \, r \quad = \quad \text{return} \, ()
\]

\[
\text{aft} \, x \, r \quad = \quad \text{do} \{(m \leftarrow \text{get}; \text{put} \, (\text{insert} \, x \, r \, m))\}
\]

\[
\text{rep} \, x \quad = \quad \text{do} \{(m \leftarrow \text{get}; \text{return} \, (m \, ! \, x))\}
\]

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. [32] 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.

**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 [28] and a corresponding conversion function to plain advice:

**type** NIAdvice a b t = \forall m. (Monad m, Monad (t m))

\[
\Rightarrow \text{Monad} \, a \, b \, (m) \, m
\]

\[
\text{niadvice} \equiv (\text{Monad} \, m, \text{MonadTrans} \, t, \text{Monad} \, (t \, m))
\]

\[
\Rightarrow \text{Monad} \, a \, b \, (m) \, m
\]

\[
\text{niadvice} \equiv \text{adv}
\]

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

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

**type** NIBase a b m = \forall t. (MonadTrans t, Monad (t m))

\[
\Rightarrow \text{Monad} \, a \, b \, (m) \, m
\]

\[
\text{nibase} \equiv (\text{Monad} \, m, \text{MonadTrans} \, t, \text{Monad} \, (t \, m))
\]

\[
\Rightarrow \text{Monad} \, a \, b \, (m) \, m
\]

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

Fortunately, the type checker confronts us with this issue by rejecting niadvice (augment (log, "eval")) and niadvice 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:

\[
\text{log}_2 \equiv (\text{Show} \, a, \text{Show} \, b) \Rightarrow \text{NIAdvice} \, a \, b \, (\text{WriterT} \, \text{String})
\]

\[
\text{beval}_1 \equiv \text{NIbase} \, \text{Expr} \, \text{Int} \, (\text{State} \, \text{Env})
\]

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

\[
\text{adv} \odot \text{bse} = \text{niadvice} \, \text{adv} \odot \text{niadvice} \, \text{bse}
\]

\[
\text{adv} \odot \text{bse} = \text{adv} \odot \text{niadvice} \, \text{bse}
\]

\[
\text{adv} \odot \text{bse} = \text{adv} \odot \text{beval}
\]

Note that, unlike Rinard’s categories, these combinators are not specific for state: they are parametric in the type of effect. The combinators \odot and \otimes closely correspond to Rinard’s interference and orthogonal categories. The \odot and \otimes 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 \(\log_2 \odot \text{beval}_1\) expresses that the logging advice and the monadic evaluator do not interfere with each other’s effects.

**Stateful Effects** Rinard et al. [32] 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 variable m. For this purpose EffectiveAdvice refines MonadState to cater for different views:

\[
\text{class} \, \text{Monad} \, m \Rightarrow \text{MonadState} \, s \, m \mid m \rightarrow s \, \text{where}
\]

\[
\text{get} :: m \, s
\]

\[
\text{class} \, \text{Monad} \, m \Rightarrow \text{MPut} \, s \, m \mid m \rightarrow s \, \text{where}
\]

\[
\text{put} : s \rightarrow m \, ()
\]

\[
\text{class} \, (\text{MonadState} \, s \, m, \text{MPut} \, s \, m) \Rightarrow \text{MonadState} \, s \, m
\]

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

\[
\text{get} \gg m \equiv m
\]

\[
\text{get} \gg \lambda s \_ \rightarrow \text{get} \gg f \, s \equiv \text{get} \gg \lambda s \_ \rightarrow f \, s \, s
\]

\[
\text{put} \, x \gg \text{put} \, y \equiv \text{put} \, y
\]

\[
\text{put} \, x \gg \text{get} \equiv \text{put} \, x \gg \text{return} \, x
\]

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

The new classes allow more accurate types, for instance dumping advice only requires reading the state:
dump₂ :: (MGet s m, MonadWriter String m, Show s) → a → m ()

With the two new constraints, EffectiveAdvice also defines relaxed versions of NIAdvice:

**type** ROAdvice a b t s = ∀m.(MGet s m, MGet s (t m)) → Open (a → t m b)

**type** WOAAdvice a b t s = ∀m.(MPut s m, MPut s (t m)) → Open (a → t m b)

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

observation :: (MGet s m, MGet s (t m), MonadTrans t) → ROAdvice a b t s → NIBase a b m → Open (a → t m b)

actuation :: (MPut s m, MPut s (t m), MonadTrans t) → WOAAdvice a b t s → NIBase a b m → Open (a → t m b)

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

bse :: ∀t.(MonadTrans t, Monad (t κ)) ⇒ Open (a → t κ β) → adv :: ∀m.(Monad m, Monad (τ m)) ⇒ Augment α β γ (τ m) where κ is a monad and τ a monad transformer. If a function proj :: ∀m, aMonad m ⇒ τ m a → m a exists that satisfies the property:

\[
\text{proj } \circ \text{lift } \equiv \text{id}
\]

, then advice adv is harmless with respect to bse:

\[
\text{proj } \circ (\text{weave } (\text{adv } \oplus \text{bse})) \equiv \text{runIdT } \circ (\text{weave } \text{bse})
\]

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 lift id expresses the intuition that projection has no impact if there are no effects.

This theorem is proved in the companion technical report [26]. 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 [36] allows the derivation of theorems for a whole class of programs, only knowing their type. Voigtlander [35] 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:

\[ \text{projW} :: \forall w \forall m a. (\text{Monad m, Monoid w}) \Rightarrow \text{WriterT} w m a \rightarrow m a \]

\[ \text{projW} m = \text{runWriterT} m \Rightarrow \text{return} \circ \text{fst} \]

It is indeed suitable:

Lemma 1 The function \( \text{projW} \) is a suitable function for the Harmless Advice theorem:

\[ \text{projW} \circ \text{lift} \equiv \text{id} \]

With the help of \( \text{projW} \), the Harmless Advice theorem establishes that the logging advice is harmless:

\[ \text{proj} \circ \text{weave} (\log_2 \text{"eval"} \oplus \text{beval}) \equiv \text{runIdT} \circ \text{weave} \circ \text{beval} \]

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

\[ \text{projS} :: \forall s m a. (\text{Monad m}) \Rightarrow s \rightarrow \text{StateT} s m a \rightarrow m a \]

\[ \text{projS} s0 m = \text{runStateT} s m s0 \Rightarrow \text{return} \circ \text{fst} \]

Indeed, the required property holds:

Lemma 2 The function \( \text{projS} \) is a suitable function for the Harmless Advice theorem:

\[ \text{projS} \circ \text{lift} \equiv \text{id} \]

for any \( s0 \).

The proofs for both lemmas are presented in the companion technical report [26].

Other Harmless Effects There are several other harmless effects, such as \( \text{IdT} \) with trivial projection function \( \text{runIdT}, \text{ReaderT} \) 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 \( \text{proj exists} \), may be used in harmless advice.

Consider again the ErrorT \( e \) monad transformer of Figure 2. We can only partially define the projection function:

\[ \text{projE} :: \forall e m a. (\text{Monad m}) \Rightarrow \text{ErrorT} e m a \rightarrow m a \]

\[ \text{projE} m = \text{runErrorT} m \Rightarrow \lambda x \rightarrow \text{case x of} \]

\[ \text{Left e} \rightarrow {?} \]

\[ \text{Right x} \rightarrow \text{return x} \]

In the case of an error, we cannot produce a value. We could attempt to fix this issue by parametrizing \( \text{projE} \) with a default value \( d \):

\[ \text{projE}' :: \forall e m a. (\text{Monad m}) \Rightarrow e \rightarrow \text{ErrorT} e m a \rightarrow m a \]

\[ \text{projE}' d m = \text{runErrorT} m \Rightarrow \lambda x \rightarrow \text{case x of} \]

\[ \text{Left e} \rightarrow \text{return d} \]

\[ \text{Right x} \rightarrow \text{return x} \]

but now \( \text{projE'} \) \( d \) :: \( \forall e m. \text{Monad m} \Rightarrow \text{ErrorT} e m a \rightarrow m a \) 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 \( \text{IO} \) monad. Only more disciplined effects, such as \( \text{WriterT}, \text{ReaderT} \) and \( \text{StateT} \) are possible.

5.4 Harmless Observation Advice

In the main Harmless Advice theorem, we have used the @ 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:

\[ \text{type NIOAugment a b c s t} = \forall m. \]

\[ (\text{MGet s m, Monad (t m)}) \Rightarrow \text{Augment a b c (t m)} \]

\[ \text{(}\circ\text{)} : (\text{MGet s m, MonadTrans t, MGet s (t m)}) \Rightarrow \text{NIOAugment a b c s t} \rightarrow \text{NILBase a b m} \]

\[ \text{Open (a \rightarrow t m b)} \]

\[ \text{adv} \circ \text{bse} = \text{augment} \circ \text{adv} \circ \text{observation} \circ \text{bse} \]

Now we can adapt the theorem accordingly:

Theorem 2 (Harmless Observation Advice) Consider any base program and any advice with the types:

\[ \text{bse :: \forall t. MonadTrans t \Rightarrow Open (a \rightarrow t \kappa \beta)} \]

\[ \text{adv :: \forall m.MonadTrans t \Rightarrow Augment a \beta \gamma (t m)} \]

\[ \text{with} \ \kappa \ \text{a MonadStatex and} \ \tau \ \text{a MonadTrans. If a function} \]

\[ \text{proj :: \forall m a. Monad m \Rightarrow} \tau \ m a \rightarrow m a \text{ exists that satisfies the property:} \]

\[ \text{proj} \circ \text{lift} \equiv \text{id} \]

, then advice \( \text{adv} \) is harmless with respect to \( \text{bse} :: \forall m. \text{Monad m} \Rightarrow \tau \ m a \rightarrow m a \) exists that satisfies the property:

\[ \text{proj} \circ \text{lift} \equiv \text{id} \]

6. LANGUAGE SUPPORT

This section discusses language support for EffectiveAdvice, including how to solve some of the current limitations.

Object-Oriented Languages The EffectiveAdvice model is easily implemented in Haskell, which is directly based on a
variant of System F. However, while Haskell provides a great setting for reasoning, it has practical drawbacks. For instance, components are not oblivious, but need to be marked Open to allow for advice.

As it turns out, some object-oriented languages like Scala, provide good linguistic support for implicit advice points\(^3\) directly based on our semantic model of advice:

- Scala supports mixins natively. Hence, Scala classes are open to mixins by default, and the native support for inheritance avoids explicit arguments like proceed. So, unlike Haskell, programs are oblivious of advice.

- 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, Scala has some support for monads [25].

Figure 10 illustrates the native support for mixins on a simple logging example for methods named get. Two unrelated classes A and B define a get method that retrieves a value stored in the corresponding objects. Note that the classes are written in the usual OO way and have not received any explicit preparation for advice. The IGet [T] trait is an interface for classes that contain a get method. The LoggedGet [T] trait implements IGet [T] by adding logging information. Mixin inheritance is used by marking the definition with abstract override, which allows the call of super.get even though get is abstract in the supertype. Note that with conventional inheritance (as in Java), this is not possible and a compile-time error is reported; the super reference corresponds to the proceed argument used in the Haskell model. Finally, the objects loggedA and loggedB provide logged implementations of the methods get for the classes A and B in an oblivious way, that is, without requiring any modifications on the original classes.

To summarize, languages that support mixin inheritance natively already provide linguistic support to the model of advice proposed in EffectiveAdvice. This support is closer to traditional AOP advice in the sense that obliviousness is preserved. A drawback of using Scala is that the theoretical developments presented in Section 5 regarding interference are not enforceable by the language. In the future we would like to combine the advantages of Haskell in terms of reasoning with the advantages of Scala for practical programming.

**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 cross-cutting 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. For example, in a big program with lots of classes like A and B in Figure 10, defining every single advice composition explicitly would be extremely tedious and difficult.

\(^3\)Note that explicit composition is still needed.

to maintain. 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.

**Obliviousness and Explicit Effects** EffectiveAdvice promotes the idea that effects should be an integral part of the interfaces of components. Adding information about effects causes some loss of obliviousness because the component needs to be written with potential effects in mind. Yet, we argue that 1) this loss is not too severe, and that 2) the benefits are quite substantial. Firstly, while it is true that we need to be conscious of effects, programs can still be written without anticipating the specific effects added by potential advice. In other words, a form of effect obliviousness exists in our approach. Secondly, the big advantage of being more conscious about effects is that modular reasoning and reasoning about interference becomes possible, using well-established reasoning techniques such as equational reasoning and parametricity. Therefore we gain quite a bit in terms of program understanding and reasoning at a relatively small cost of being more explicit about effects.

**Explicit effects in practice** A related question concerns the practical use of explicit effects. In purely functional languages like Haskell, explicit effects are the only kinds of effects, but most programmers (not using purely functional programming) are used to implicit effects. For example, the Scala advice in Figure 10 makes use of implicit side-effects, which is inline with what a typical programmer would write, but goes against our premise of making effects explicit. It is possible to program in Scala with explicit effects. The technical report [26] shows one alternative way to program advice in Scala using monads. However, the question remains whether programmers accept monads. Peyton Jones and Wadler [27] show that the monadic programming style is quite close to imperative programming. Yet, we acknowledge that the monadic style can be hard to grasp at times, and more work is needed to make particular situations more manageable. For instance, using different instances of the same monad transformer within different components is very awkward. This may arise when combining different advices on the same base components, or when using multiple advised components within the same application.

**Type Discipline** Our type-based approach conservatively approximates harmless advice. Some advice implementations are harmless, even though they do not have the appropriate declared type or have not written in the right form for our approach. Then program transformations based on equational reasoning may help to expose the appropriate form and type. At other times, the advice is harmless only conditionally, when used in particular restricted circumstances. For instance, the memoization advice is harmless when used with effect-free base programs like the fibonacci function. In such cases, conditional harmless theorems may depend on the actual implementations.

7. RELATED WORK

Kiczales et al. [17] 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 [13]: the distinguishing characteristics of AOP systems are support for quantification and obliviousness. Quantification is the ability to write sep-
arate 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.

**Functional AOP systems** Two main approaches to functional AOP exist, both following the pointcut-advice model: 1) *statically typed language-based* approaches such as AspectCaml [23], AspectFun [4] and AspectML [9], and 2) *lightweight dynamically typed* approaches like AspectScheme [12]. 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 [14] and expressing dynamically deployed aspects [34]. 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, “features” (such as first-class, polymorphic and inferable types for advice) come for free. In language-based approaches adding support for each of these features is non-trivial, and only AspectML supports all of them.

**Mixins** Filman and Friedman argue that many systems supporting a form of mixin inheritance [7, 2] 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. Mixin inheritance as been widely used in functional programming [7, 24, 18, 3], using techniques similar to that in Section 2. However, only Brown and Cook [3] have used it with explicit effects, to modularize memoization.

**Modular Reasoning** Kiczales and Mezini [16] argue that modular reasoning about cross-cutting aspects is impossible. 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 [1] does define the concept of Open Modules that allows modular reasoning. However, this approach is severely limited: reasoning about 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 interference as an important factor in reasoning about advice.

In the context of POP Prehofer [31] defines a notion similar to harmless advice, but with two important differences. Firstly in Prehofer’s monadic model there is no use of open recursion, which makes it hard to model tightly coupled advice such as memoization. Secondly, the approach used to reason about harmlessness is quite different. Instead of using parametricity, Prehofer requires a certain syntactic pattern for his form of harmless advice. Exploiting this syntactic pattern it is possible to reason by induction on the operation sequences and equational reasoning to prove a harmless advice-like theorem. This also allows for conditional harmlessness, that is not covered by our approach. An advantage of our approach is that any piece of advice of a certain type can be proved harmless once and for all by applying our harmlessness theorems. In contrast, with conservative redefinitions each composition of advice needs to be checked for harmlessness. Nevertheless, we believe that given sufficiently polymorphic advice it should be possible to use parametricity in Prehofer’s setting to prove that a piece of advice is conservative regardless of the base program.

Prehofer [29] also considers when the composition of two conservative extensions is conservative: not always, because the form of composition depends in an ad-hoc manner on the involved advices. Using our approach, the uniformity of composition seems to suggest that the composition of two harmless advices is always harmless, but this needs further investigation.

Dantas and Walker [8] propose a type-and-effect system for identifying harmless advice on the MinAML core language [21]: protection domains prevent information flow from advice to base program. Their modular analysis supports a formal result similar to our harmless observation advice theorem. Orthogonal data flow interference cannot be enforced, and it is not clear how non-stateful effects like exceptions fit in their approach. Because MinAML is impure, effects are needed in addition to types.

Clifton and Leavens [5] identify that observers (harmless observation advice) do not change the specification of the advised module. Later, Clifton et al. [6] propose an extension of AspectJ with (optional) annotations for control and heap effects, which are similar to Rinard et al.’s two forms of interference. A type-and-effect system is used to modularly verify the annotations. Spectator advice is their counterpart of harmless observation advice, and they prove that it does not modify the base program’s state. No formal statement is made about the lack of control-flow interference.

Douence et al. [11] present a formal approach for determining strong independence of stateful aspects: when aspects commute, they do not interfere with each other. Equational reasoning laws are used to determine (non-modularly) whether two given aspect implementations commute; in contrast, EffectiveAdvice only looks at the types. No formal statement is provided. They focus on aspect/aspect interaction and overlapping pointcuts, and do not address aspect/base program interaction. Moreover, the insert language is only partially defined and no equational reasoning laws for effects are provided. While this paper focuses on the advice/base program interference, the same approach applies equally to the interaction of two aspects.

Rinard et al. [32] formulate a classification scheme for different forms of interference, and combine a number of program analyses for automated classification. No formal results are proved.

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 light-weight 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 and modularly about programs using familiar techniques such as equational reasoning and parametricity.

**Aspects and Effects** The connection between AOP and effects is a recurring theme since De Meuter [10] argued for the use of monads as a theoretical foundation for AOP: this view is not widely accepted. Hofer and Ostermann [15] argued recently that “monads and aspects have to be regarded as quite different mechanisms”. EffectiveAdvice shows that aspects (when understood as advice) and monads have complementary roles when it comes to separation of concerns:
aspects provide textual separation of code, and monads provide conceptual separation of effects used by different aspects. The relationship between aspects and effects is not 1-to-1, since an aspect may produce several types of effects, and the same effect may be manipulated by several aspects.

8. CONCLUSION

EffectiveAdvice promotes the idea that effects should be an integral part of the interface of components, avoiding hidden data flows. 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 is possible by looking at the interfaces only.

EffectiveAdvice provides a simple and lightweight model of advice that can be elaborated as a Haskell library. Some languages like Scala already provide linguistic support for an advice-like mechanism that is based directly in our model of advice, although implicit data flows cannot be ruled out. Our hope is that EffectiveAdvice brings new insights that help designing new languages aimed at addressing the problem of crosscutting concerns while, at the same time, supporting nice modular and interference reasoning properties.

Acknowledgments

We thank J. Aldrich, B. Delaware, M. van Dooren, J. Gibbons, S. Peyton Jones, S. Krishnamurthi, A. Moors, J. Voigtlander and M. Wang for their comments; and A. Löh for support.

William R. Cook was supported by the NSF grants CCF-0448128, CCF-0724979 and CCF-6752487.

References