Under consideration for publication in J. Functional Programming

MRI: Modular Reasoning about Interference in Incremental Programming

BRUNO C. D. S. OLIVEIRA
National University of Singapore
TOM SCHRIJVERS
Dept. of Applied Mathematics and Computer Science
Ghent University, Belgium
WILLIAM R. COOK
University of Texas at Austin, USA

Abstract

Incremental Programming (IP) is a programming style in which new program components are defined as increments of other components. Examples of IP mechanisms include: *Object-oriented programming* (OOP) inheritance, *aspect-oriented programming* (AOP) advice and *feature-oriented programming* (FOP). A characteristic of IP mechanisms is that, while individual components can be independently defined, the composition of components makes those components become tightly coupled, sharing both control and data flows. This makes reasoning about IP mechanisms a notoriously hard problem: *modular reasoning* about a component becomes very difficult; and it is very hard to tell if two tightly coupled components *interfere* with each other's control and data flows.

This paper presents *modular reasoning about interference* (MRI), a *purely functional* model of IP embedded in Haskell. MRI models inheritance with mixins and side-effects with monads. It comes with a range of powerful reasoning techniques: equational reasoning, parametricity and reasoning with algebraic laws about effectful operations. These techniques enable modular reasoning about interference in the presence of side-effects.

MRI formally captures *harmlessness*, a hard-to-formalize notion in the interference literature, in two theorems. We prove these theorems with a non-trivial combination of all three reasoning techniques.

1 Introduction

Cook and Palsberg (1989) define *Incremental Programming* (IP) as a programming style in which new program components are defined as increments of existing ones. *Object-oriented programming* (OOP) inheritance (Dahl & Nygaard, 1966) is probably the most widely used mechanism for IP; other specialized forms of IP include variants of inheritance such as *mixins* (Bracha & Cook, 1990; Flatt *et al.*, 1998); *aspect-oriented programming* (AOP) (Kiczales *et al.*, 1997) and *feature-oriented programming* (FOP) (Prehofer, 1997).

A characteristic of IP systems is that the control and data flows between the derived and the original components are quite complex, since control flows back and forth between components in a composition. In other words, despite being textually separated, IP components are semantically coupled. 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 has been intensely studied in the OOP and AOP literature (Stata & Guttag, 1995; Kiczales & 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 & Walker, 2006; Clifton *et al.*, 2007; Bagherzadeh *et al.*, 2011); and the OOP literature addresses symptoms of interference (Clifton *et al.*, 2007), but is perhaps less aware of the general notion.

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.

Because IP systems share similar characteristics and problems, it is reasonable to expect that there is a general framework which can describe those systems. The benefits of such general framework is that once it is shown that a certain property holds in the general framework, the fact that that property holds for particular instances comes for free.

One such framework is the *denotational model* of inheritance proposed by Cook (1989), which serves as the starting point for this article. This model uses traditional techniques of *fixed-point theory*, allowing inheritance to be understood compositionally. One advantage is that modular reasoning is to an extent possible since no effects are considered. In other words, the setting is a variation of a pure lambda calculus in which *equational reasoning* is possible. The benefits of purely functional settings for modular reasoning about forms of IP have been explored in more detail by other authors. Most prominently, *Open Modules* (Aldrich, 2005) support reasoning about tightly coupled components. In Open Modules, effects are not considered and modular reasoning about components is possible through *logical relations*, which play a similar role to *equational reasoning* in more conventional purely functional languages. By not allowing any effects, the biggest obstacle to reasoning is removed. Unfortunately, this solution is not effective in practice, as almost all practical uses of IP involve effects.

The latter observation leads us to the principal goal of this article, which concerns answering the question of whether it is possible to have a model of IP with effects, while supporting both modular reasoning *and* reasoning about non-interference of effects. To our knowledge there is no IP approach capable of handling both reasoning concerns at the same time. This is understandable because the introduction of *implicit effects* is well-known to destroy the nice reasoning properties of pure languages. It thus seems that we have a dilemma: on the one hand it is possible to have a pure language with modular reasoning properties, but in which most interesting uses of IP are not possible; and on the other hand it is possible to have an impure language in which interesting uses of IP are possible, but modular reasoning is significantly undermined. Indeed this has lead some authors, including Kiczales and Mezini (2005), to argue that modular reasoning about mechanisms that capture crosscutting concerns (a particular yet significant case of IP) is simply not possible, and that a degree of global analysis is always needed.

Inspired by research on handling effects in purely functional languages, we propose *modular reasoning about interference* (MRI) in incremental programming as an extension of Cook's semantic model of inheritance with *explicit effects* through the use of *monads* (Wadler, 1992a). This enables uses of IP involving effects without losing the purity and the reasoning properties of the model. We do not devise a novel core language, but reuse the well-studied *polymorphic λ-calculus*, System F_ω (Reynolds, 1974), extended with recursion and benefit from the many established technical results. Like other authors (Wadler, 1989; Voigtländer, 2009), we use Haskell as a convenient source language for System F_ω and elaborate the model as a Haskell library[1].

Our model is well-suited to make formal statements about non-interference. Inspired by Dantas and Walker's (2006) notion of *harmless advice*, we express two non-trivial theorems about harmless mixin inheritance. This is possible because of the purely functional nature of our setting, which provides us with a range of powerful techniques to reason about such formal statements: *equational reasoning*, *parametricity* and reasoning with *algebraic laws about (monadic) effectful operations*.

MRI is a compelling application of the latter two techniques, which are particularly relevant for reasoning about effectful programs, but have been relatively unexplored so far. Most importantly, these techniques allow powerful forms of *modular reasoning*: parametricity enables reasoning based on types only and not the implementation of components, while algebraic laws support reasoning independent of the implementation details of effects (monads). With respect to parametricity our work builds on foundational work by Voigtländer (2009) and shows how parametric properties about effectful programs are important to state basic non-interference properties. With respect to algebraic laws our proof for the *harmless observation mixin* theorem (Oliveira *et al.*, 2010) (see also Section 5.2) provides an application of algebraic laws for stateful monadic effects. Interestingly, while work on reasoning about pure functional programs abounds, there is far less work on reasoning about effectful monadic programs using algebraic laws. One notable exception is the work by Liang and Hudak (1996), where they have shown how to reason about monadic programs with laws for the reader monad. With respect to other types of effects, and as far as we can tell, the laws about stateful monadic effects presented in Oliveira et al. (2010) have not appeared before in the literature. More recently Gibbons and Hinze (2011) picked up on this topic and have explored algebraic properties for various types of monadic effects.

In summary, the contributions of this paper are:

- MRI: a purely functional model for incremental programming with effects (see Section 2). Monads make effects an integral part of each component's interface. Thus, MRI allows interesting, effectful, programs to be expressed, while still supporting all the benefits of purely functional programming. We illustrate our model on an interpreter, modularizing orthogonal aspects of computation such as logging and tracing through mixin inheritance.
- A non-trivial application of various functional programming reasoning techniques to the notoriously hard problem of modular reasoning about inheritance in the presence of effects. Our approach combines familiar reasoning techniques such as *equational reasoning* and *parametricity* (see Sections 3 and 4) with some relatively unexplored techniques to reason about effectful code. These techniques are used to prove two harmless mixin theorems (see Section 5).

[1] `http://users.ugent.be/~tschrijv/MRI/`

- Two different techniques to reason about non-interference of mixin components. We first present a simple, but non-modular approach in Section 3. This technique allows us to reason about any non-interference of two individual components and proofs can be easily mechanized in theorem provers like Coq. We then present a modular and more generic approach in Section 5. This approach allows general non-interference statements, for any given mixin and base programs, provided that they conform to a suitable type scheme and are composed with an appropriate mixin combinator.
- A classification system for mixin interference inspired by Rinard et al.'s (2004) similar classification for AOP advice (see Section 4). We adapt Rinard et al.'s control and data flow classification for advice to mixins and we provide a fine-grained classification for stateful effects.
- An implementation of the MRI model as a Haskell library using open recursion to model mixin inheritance and monads to model effects. The model is *statically typed* and *purely functional*.

We believe that these results are relevant to the common problems of all IP approaches and provide strong incentives for OOP and other IP instances based on inheritance to consider similar solutions.

This paper is an extended version of a paper published at AOSD (Oliveira *et al.*, 2010). With respect to that paper we generalize the scope of our work from AOP-style advice to inheritance, elaborate on the proof techniques used and include the proofs of our theorems. Furthermore the technique to reason about non-interference presented in Section 3 is new. While this approach is non-modular, it has the advantage that it can be used to prove some harmlessness results that are not possible to prove with the modular technique presented in Section 5. For example, the proof that memoization does not interfere with the Fibonacci function (see Section 3) does not follow from the harmless mixin theorems, but can be proved directly with our new technique.

2 MRI

This section introduces the Haskell implementation of MRI using open recursion and monads. Monads and monad transformers are introduced briefly in Section 2.2, but more thorough introductions can be found in the literature (Wadler, 1992a; Liang *et al.*, 1995).

2.1 Open Recursion

The standard mechanism for extending a component's behavior is through composition or decoration. However, this only affects the external clients of the extended component, and not the internal *recursive* uses. Open recursion is a way of structuring a component that leaves recursive references open, so that the recursive behavior can be extended too. This is the basis for modeling inheritance and mixin composition in object-oriented languages (Cook, 1989), and it also provides a simple model for some types of aspects (Lopez-Herrejon *et al.*, 2006; Oliveira *et al.*, 2010).

We now present the open recursive model of inheritance, formulated in Haskell, that is used throughout this article.

```
type Open s = s → s

new :: Open s → s
new a = let this = a this in this

zero :: Open s
zero = id

(⊕) :: Open s → Open s → Open s
a1 ⊕ a2 = λsuper → a1 (a2 super) super
```

Fig. 1. Basic inheritance model.

Inheritance Model Schematically, our denotational model of inheritance represents the composition of components with open recursion as follows:

$$\begin{tikzcd}
p = new\ (a_1 \arrow[rr, bend right, "super"'] & \oplus & a_2 \arrow[rr, bend right, "super"'] & \oplus & \ldots \arrow[rr, bend right, "super"'] & \oplus & a_n \arrow[rr, bend right, "super"'] & \oplus & base) \arrow[llllllll, bend right, "this"']
\end{tikzcd}$$

The open *base* component provides base behavior similar to a base class, and the other *mixin* components $a_1, \ldots, a_n$ provide behavior extensions, similar to AOP advice or Scala's mixins. The inheritance operator $\oplus$ extends one component with another; extensions are applied from right to left. Finally, the *new* operator closes an open component; using OOP terminology, this operator instantiates an object p of the class $a_1 \oplus a_2 \oplus \ldots \oplus a_n \oplus base$.

The arrows in the diagram show what happens to the references during composition. The $\oplus$ operator instantiates the `super` reference of the extending component with the extended component. In contrast, the *new* operator instantiates the self-reference (`this`) of the base component to the entire composition.

The basis of the implementation is shown in Figure 1. The type *Open s* is a synonym for a function of type $s \rightarrow s$ representing an open component of type s. The parameter s of this function is the self-reference and the return value is the resulting closed module. The inheritance operator $\oplus$ defines component composition. Composition is associative, and it has the *zero* component as left and right unit, forming a monoid.[2]

$$\begin{aligned} f \oplus zero \;\equiv\; f \;&\equiv\; zero \oplus f \\ (f \oplus g) \oplus h \;&\equiv\; f \oplus (g \oplus h) \end{aligned}$$

The function *new* is a fixpoint combinator used for closing, or instantiating, an open and potentially extended component.

There are several other models of inheritance. Cook (1989) explores many other variants. We believe that similar results to those in this paper can be obtained for these other models.

We adopt the model of inheritance in Figure 1 because it is simple yet expressive enough to tackle the reasoning issues at the heart of this paper. In this model, which is polymorphic

[2] *Open s* is the monoid of endofunctions with identity and function composition; $\equiv$ means denotational equivalence throughout the article.

in the type *s* of open modules, we can capture the simplest form of open modules that is still interesting: single-argument functions.

Example The open (single-argument) function fib_1 defines the standard Fibonacci function, except that recursive calls are replaced by *this*.

```
fib₁ :: Open (Int → Int)
fib₁ this n = case n of 0 → 0
                        1 → 1
                        _ → this (n − 1) + this (n − 2)
```

The open function *optfib* optimizes two calls of the Fibonacci function by returning the appropriate values immediately. Note that *optfib* 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.

```
optfib :: Open (Int → Int)
optfib super n = case n of 10 → 55
                           30 → 832040
                           _  → super n
```

Different combinations of open functions are closed with *new*:

```
slowfib₁, fastfib₁ :: Int → Int
slowfib₁ = new fib₁
fastfib₁ = new (optfib ⊕ fib₁)
```

The functions $slowfib_1$ and $fastfib_1$ illustrate that MRI unifies the concept of extensions and base programs under a single type. There is still a conceptual difference between them, because in a base program *this* is understood as a recursive call, while in the mixin *super* refers to the original computation that is extended by that mixin. Instantiating extensions alone will typically result in a useless program, as it has no base case.

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

2.2 Monads and Monad Transformers

Monads are a standard technique for encapsulating computational effects in pure functional languages (Wadler, 1992a). 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
```

$$
\begin{array}{lll}
\multicolumn{3}{l}{\quad\text{-- Identity}} \\
\mathit{run}\mathbb{I} & :: \mathbb{I}\ a & \rightarrow a \\
\mathit{run}\mathbb{I}_{\mathrm{T}} & :: \mathbb{I}_{\mathrm{T}}\ m\ a & \rightarrow a \\
\multicolumn{3}{l}{\quad\text{-- State}} \\
\mathit{run}\mathbb{S} & :: \mathbb{S}\ s\ a & \rightarrow s \rightarrow (a, s) \\
\mathit{eval}\mathbb{S} & :: \mathbb{S}\ s\ a & \rightarrow s \rightarrow a \\
\mathit{run}\mathbb{S}_{\mathrm{T}} & :: \mathbb{S}_{\mathrm{T}}\ s\ m\ a & \rightarrow s \rightarrow m\ (a, s) \\
\multicolumn{3}{l}{\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathbb{S}_{\mathrm{M}}\ s\ m \mid m \rightarrow s\ \textbf{where}} \\
\multicolumn{3}{l}{\quad \mathit{get} :: m\ s} \\
\multicolumn{3}{l}{\quad \mathit{put} :: s \rightarrow m\ ()} \\
\multicolumn{3}{l}{\quad\text{-- Writer}} \\
\mathit{run}\mathbb{W} & :: \mathbb{W}\ w\ a & \rightarrow (a, w) \\
\mathit{eval}\mathbb{W} & :: \mathbb{W}\ w\ a & \rightarrow a \\
\mathit{exec}\mathbb{W} & :: \mathbb{W}\ w\ a & \rightarrow w \\
\mathit{run}\mathbb{W}_{\mathrm{T}} & :: \mathbb{W}_{\mathrm{T}}\ w\ m\ a & \rightarrow m\ (a, w) \\
\mathit{eval}\mathbb{W}_{\mathrm{T}} & :: \mathbb{W}_{\mathrm{T}}\ w\ a & \rightarrow a \\
\mathit{exec}\mathbb{W}_{\mathrm{T}} & :: \mathbb{W}_{\mathrm{T}}\ w\ a & \rightarrow w \\
\multicolumn{3}{l}{\textbf{class}\ (\mathit{Monoid}\ w, \mathit{Monad}\ m) \Rightarrow \mathbb{W}_{\mathrm{M}}\ w\ m \mid m \rightarrow w\ \textbf{where}} \\
\multicolumn{3}{l}{\quad \mathit{tell} :: w \rightarrow m\ ()} \\
\multicolumn{3}{l}{\quad\text{-- Reader}} \\
\mathit{run}\mathbb{R} & :: \mathbb{R}\ e\ a & \rightarrow e \rightarrow a \\
\mathit{run}\mathbb{R}_{\mathrm{T}} & :: \mathbb{R}_{\mathrm{T}}\ e\ m\ a & \rightarrow e \rightarrow m\ a \\
\multicolumn{3}{l}{\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathbb{R}_{\mathrm{M}}\ e\ m \mid m \rightarrow e\ \textbf{where}} \\
\multicolumn{3}{l}{\quad \mathit{ask} :: m\ e} \\
\multicolumn{3}{l}{\quad\text{-- Error}} \\
\mathit{run}\mathbb{E} & :: \mathit{Error}\ e\ a & \rightarrow \mathit{Either}\ e\ a \\
\mathit{run}\mathbb{E}_{\mathrm{T}} & :: \mathbb{E}_{\mathrm{T}}\ e\ m\ a & \rightarrow m\ (\mathit{Either}\ e\ a) \\
\multicolumn{3}{l}{\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathbb{E}_{\mathrm{M}}\ e\ m \mid m \rightarrow e\ \textbf{where}} \\
\multicolumn{3}{l}{\quad \mathit{throwError} :: e \rightarrow m\ a} \\
\multicolumn{3}{l}{\quad \mathit{catchError} :: m\ a \rightarrow (e \rightarrow m\ a) \rightarrow m\ a}
\end{array}
$$

Fig. 2. Monads and monad transformer types.

The type $m\ a$ describes computations of type m 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. The convenient funtion $\gg$ defines a special case of bind where the intermediate value is not used:

$$
\begin{array}{l}
(\gg) :: \mathit{Monad}\ m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b \\
\mathit{ma} \gg \mathit{mb} = \mathit{ma} \gg\!= \backslash\_ \rightarrow \mathit{mb}
\end{array}
$$

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

Definition 1 (MONAD LAWS)

$$\begin{array}{rcll} \mathit{return}\ x \gg\!= f & \equiv & f\ x & \text{(RETURN-BIND)} \\ p \gg\!= \mathit{return} & \equiv & p & \text{(BIND-RETURN)} \\ (p \gg\!= f) \gg\!= g & \equiv & p \gg\!= \lambda x \to (f\ x \gg\!= g) & \text{(BIND-BIND)} \end{array}$$

The Haskell **do** notation is syntactic sugar for the *bind* operator: **do** $\{x \leftarrow f; g\}$ means $f \gg\!= \lambda x \to g$.

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*. For each particular type of effect (such as state or exceptions) there is an associated monad transformer type and type class. Figure 2 shows a number of monad and monad transformer (Liang *et al.*, 1995) definitions that are used throughout the paper. Note that classes such as $\mathbb{S}_{\mathrm{M}}\ s\ m$ use *functional dependencies* (Jones, 2000). The annotation $m \to s$ states that there is a functional dependency between the type *m* and *s* (the type *m* determines the type *s*). This additional information is used by the compiler to improve type-inference.

2.3 Monadic Mixins

The combination of mixins and monads is the key to provide a purely functional model of IP with effects. For practical applications, pure mixins are of limited use. For instance, most well-known examples of AOP advice, including logging, tracing, backups, and memoization, are effectful. A setting without effects is severely limited. Take the example in Section 2.1. Ideally it should be possible to dynamically construct a 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 mixins are 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 mixin is presented in Figure 3. The $\mathbb{S}_{\mathrm{M}}$ class, which models state, is used by the *memo* mixin to read and update the cached values in the memo table. The memo table is implemented using a (finite) 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 2, to recover

```
memo :: $\mathbb{S}_M$ (Map Int Int) m ⇒ Open (Int → m Int)
memo super x = do m ← get
                  if member x m then return (m ! x)
                                else do y ← super x
                                        m′ ← get
                                        put (insert x y m′)
                                        return y
fib₂ :: Monad m ⇒ Open (Int → m Int)
fib₂ this n = case n of 0 → return 0
                        1 → return 1
                        _ → do y ← this (n − 1)
                               x ← this (n − 2)
                               return (x + y)
```

Fig. 3. Memoization

```
data Expr
  = Lit Int
  | Var String
  | Plus Expr Expr
  | Assign String Expr
  | Sequence [Expr]
  | While Expr Expr
type Env = [(String, Int)]
```

Fig. 4. Types for a simple imperative language.

variations of the Fibonacci function. For example, the identity monad $\mathbb{I}$ recovers the effect-free function

$$slowfib_2 :: Int \to Int$$
$$slowfib_2 = run\mathbb{I} \circ new\ fib_2$$

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

$$eval\mathbb{S} :: \mathbb{S}\ s\ a \to s \to a$$
$$eval\mathbb{S}\ m\ s = fst\ \$\ run\mathbb{S}\ m\ s$$

$$fastfib :: Int \to Int$$
$$fastfib\ n = eval\mathbb{S}\ (new\ (memo \oplus fib_2)\ n)\ empty$$

2.4 Application: Orthogonal Aspects of Interpreters

In this section we show how monadic mixins can capture various orthogonal aspects of interpreters (and, more generally, programs) like logging or improved error handling.

Figure 4 shows the datatype for the abstract syntax of a simple imperative language. In Figure 5 the interpreter for that language is presented. The interpreter's type *Open* $(Expr \to m\ Int)$ indicates that it is an open function of type $Expr \to m\ Int$. The type variable m

```
beval :: 𝕊_M Env m ⇒ Open (Expr → m Int)
beval this 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 ← this l
                         y ← this r
                         return (x + y)
  Assign x r        → do y ← this r
                         e ← get
                         put ((x, y) : e)
                         return y
  Sequence [ ]      → return 0
  Sequence [x]      → this x
  Sequence (x : xs) → this x ≫ this (Sequence xs)
  While c b         → do x ← this c
                         if (x ≡ 0) then return 0
                           else (this b ≫ this exp)
  where msg = "Variable not found!"
```

Fig. 5. A mixin-based monadic evaluator.

means that mixins may introduce effects. However, the constraint on *m* is not *Monad m*, for an unknown type of effect, but $\mathbb{S}_{\mathrm{M}}$ *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 → 𝕊 Env Int
eval = new beval
```

The self-reference in the open function is closed, and *m* is instantiated to the state monad.

Figure 6 shows how to define various mixins that capture aspects which are orthogonal to the base computations. These mixins are described next.

Logging mixin This mixin defines logging modularly. It writes a log message when entering the function call, delegates to *super* and finally writes another log message when exiting. It uses the writer monad $\mathbb{W}_{\mathrm{M}}$ of Figure 2 for writing logging messages.

Dumping mixin This mixin shows how to modularly dump the environment at each evaluation step, which is useful for debugging. The mixin intercepts the evaluation of every expression, retrieves the current environment, writes it out using a writer monad and delegates the actual evaluation to *super*. This example is interesting because it shows that the mixin not only introduces its own writer effect $\mathbb{W}_{\mathrm{M}}$, but also relies on the presence of the state effect $\mathbb{S}_{\mathrm{M}}$.

```
  -- the logging mixin
log :: (𝕎_M String m, Show a, Show b) ⇒ String → Open (a → m b)
log name super x = do
  tell ("Entering " ++ name ++ "with" ++ show x ++ "\n")
  y ← super x
  tell ("Exiting " ++ name ++ "with" ++ show y ++ "\n")
  return y
  -- the environment dumping mixin
dump :: (𝕊_M s m, 𝕎_M String m, Show s) ⇒ Open (a → m b)
dump super arg = do s ← get
                    tell (show s ++ "\n")
                    super arg
  -- the exception handling mixin
type Exc = (String, Expr, Env)
eeval :: (𝕊_M Env m, 𝔼_M Exc m) ⇒ Open (Expr → m Int)
eeval super exp = case exp of
  Var s → do e ← get
             case lookup s e of
               Just x → return x
               _      → throwError (msg, exp, e)
  _     → super exp
  where msg = "Variable not found!"
```

Fig. 6. Logging, environment dumping and exception handling mixins.

Exception handling mixin A last example of a useful mixin is 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 mixin overrides the case for variables and replaces the *error* primitive by *throwError* (see Figure 2). 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 in the type of the evaluator, and the client code must handle the exception, which ensures that the main program remains in a usable state. As in the dumping mixin, two different types of monads are involved: state $\mathbb{S}_{\mathrm{M}}$ and error $\mathbb{E}_{\mathrm{M}}$.

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

```
debug_1, debug_2 :: (𝕎_M String m, 𝕊_M Env m) ⇒ Expr → m Int
debug_1 = new (log "eval" ⊕ beval)
debug_2 = new (log "eval" ⊕ dump ⊕ beval)

exc :: (𝔼_M Exc m, 𝕎_M String m, 𝕊_M Env m) ⇒ Expr → m Int
exc = new (eeval ⊕ log "eval" ⊕ beval)
```

The $\mathit{debug_1}$ program adds logging of function calls to the evaluator, while $\mathit{debug_2}$ 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).

$$
\begin{array}{l}
\mathit{test_1}\ e = \mathit{eval}\mathbb{S}\ (\mathit{exec}\mathbb{W}_{\mathrm{T}}\ (\mathit{debug_1}\ e))\ [\,]\\
\mathit{test_2}\ e = \mathit{eval}\mathbb{S}\ (\mathit{exec}\mathbb{W}_{\mathrm{T}}\ (\mathit{debug_2}\ e))\ [\,]\\
\mathit{test_3}\ e = \mathit{extract}\ (\mathit{eval}\mathbb{S}_{\mathrm{T}}\ (\mathit{exec}\mathbb{W}_{\mathrm{T}}\ (\mathit{exc}\ e))\ [\,])\\
\quad \textbf{where}\\
\qquad \mathit{extract}\ (\mathit{Left}\ (\mathit{msg}, \mathit{exp}, \_)) = \texttt{"Error: "} + \!\!+\ \mathit{msg} + \!\!+\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{"\textbackslash nIn Expression: "} + \!\!+\ \mathit{show}\ \mathit{exp}\\
\qquad \mathit{extract}\ (\mathit{Right}\ t) \qquad\quad\;\; = t
\end{array}
$$

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.

3 Interference and Equational Reasoning

In IP it is often the intent of the programmer that inheritance extends or augments but does not modify the behavior of a base component. This property is called non-interference. The opposite, interference, means that inheritance causes the components to interact in a way that essentially changes the behavior of the base component. Knowing whether two components interfere directly contributes to programmer understanding: If components do not interfere, then they can be understood individually. If the components do interfere, then the programmer should more carefully investigate what the impact of interference is. Moreover, in many applications, the programmer intends to implement non-interfering inheritance; interference is then a programming error. Either way, the programmer's understanding (or confidence in his understanding) is greatly improved when he can establish whether two components do or do not interfere.

3.1 Equational Reasoning about Interference

As a formal model of inheritance, MRI does allow us to reason formally about (non-) interference. This does not require special-purpose mechanisms such as Aldrich's logical equivalence laws (Aldrich, 2005). Instead, Haskell's equational reasoning allows us to establish non-interference from the equality of two expressions. Furthermore, the use of simple equational reasoning steps allows us to mechanize such non-interference proofs in theorem provers like Coq. The proofs for the two theorems presented in this section have been formalized in Coq and are available at `http://users.ugent.be/~tschrijv/MRI`.

To formally reason about non-interference, we must first be able to formally capture this often quite informal notion. MRI allows us to do so. For instance, we may want to express that the logging *log* component does not interfere with the $\mathit{fib_2}$ component. In the spirit of equational reasoning, we capture this informal statement in a formal equality:

Theorem 1 (LOGGING NON-INTERFERENCE)
For all $n > 0$, we have that

$$\mathit{eval}\mathbb{W}\ (\mathit{logfib}\ n) \equiv \mathit{run}\mathbb{I}\ (\mathit{slowfib}_2\ n)$$

where we use the following definition of *logfib*, slightly simplified for the sake of brevity:

```
logfib = new (log ⊕ fib₂)
  where
    log :: 𝕎_M String m ⇒ Open (Int → m Int)
    log super n = do tell "entering fib"
                     super n
```

This theorem relates the composition of *log* and fib_2 to fib_2 by itself. On the right-hand side, we have the pure Fibonacci function. On the left-hand side, we ignore the side-effects of the logging mixin with the help of $\mathit{eval}\mathbb{W} :: \mathbb{W}\ a \to a$, which projects a computation in the $\mathbb{W}$ monad on its return value.

Proof
The proof of this formal statement with equational reasoning proceeds by induction. The two base cases, 0 and 1, are trivial: evaluate the left- and right-hand sides and observe that they are equal. The inductive case is more interesting.

```
  eval𝕎 (logfib (n+2))
≡ {-unfold logfib and fix -}
  eval𝕎 (log ∘ fib₂ ∘ logfib (n+2))
≡ {-unfold log -}
  eval𝕎 (do tell "entering fib."
            fib₂ logfib (n+2))
≡ {-unfold fib₂ & reduce case -}
  eval𝕎 (do tell "entering fib"
            f₁ ← logfib (n+1)
            f₂ ← logfib n
            return (f₁ + f₂))
≡ {-unfold return -}
  eval𝕎 (do tell "entering fib"
            f₁ ← logfib (n+1)
            f₂ ← logfib n
            𝕎 (f₁ + f₂, ""))
≡ {-unfold >>= -}
  eval𝕎 (do tell "entering fib"
            f₁ ← logfib (n+1)
            𝕎 (eval𝕎 (𝕎 (f₁ + eval𝕎 (logfib n), ""))
               , exec𝕎 (logfib n) ++
                 exec𝕎 (𝕎 (f₁ + eval𝕎 (logfib n), ""))))
```

```
≡  {-(1) eval𝕎 (𝕎 (x,y)) ≡ x and (2) exec𝕎 (𝕎 (x,y)) ≡ y -}
  eval𝕎 (do tell "entering fib"
              f₁ ← logfib (n+1)
              𝕎 (f₁ + eval𝕎 (logfib n)
                 ,exec𝕎 (logfib n) ++ ""))
≡  {-(3) l ++ "" ≡ l -}
  eval𝕎 (do tell "entering fib"
              f₁ ← logfib (n+1)
              𝕎 (f₁ + eval𝕎 (logfib n)
                 ,exec𝕎 (logfib n)))
≡  {-unfold >>= -}
  eval𝕎 (do tell "entering fib"
              𝕎 (eval𝕎 (𝕎 (eval𝕎 (logfib (n+1)) + eval𝕎 (logfib n)
                              ,exec𝕎 (logfib n)))
                 ,exec𝕎 (logfib (n+1)) ++
                  exec𝕎 (𝕎 (eval𝕎 (logfib (n+1)) + eval𝕎 (logfib n)
                              ,exec𝕎 (logfib n)))))
≡  {-eval𝕎 (𝕎 (x,y)) ≡ x and exec𝕎 (𝕎 (x,y)) ≡ y -}
  eval𝕎 (do tell "entering fib"
              𝕎 (eval𝕎 (logfib (n+1))  +  eval𝕎 (logfib n)
                 ,exec𝕎 (logfib (n+1)) ++ exec𝕎 (logfib n)))
≡  {-unfold >>= -}
  eval𝕎 (𝕎 (eval𝕎 (𝕎 (eval𝕎 (logfib (n+1))  +  eval𝕎 (logfib n)
                          ,exec𝕎 (logfib (n+1)) ++ exec𝕎 (logfib n)))
              ,exec𝕎 (tell "entering fib.") ++
               exec𝕎 (𝕎 (eval𝕎 (logfib (n+1))  +  eval𝕎 (logfib n)
                          ,exec𝕎 (logfib (n+1)) ++ exec𝕎 (logfib n)))))
≡  {-eval𝕎 (𝕎 (x,y)) ≡ x -}
  eval𝕎 (logfib (n+1)) + eval𝕎 (logfib n)
≡  {-induction hypotheses -}
  run𝕀 (slowfib₂ (n+1)) + run𝕀 (slowfib₂ n)
≡  {-(4) run𝕀 (slowfib₂ (n+1)) + run𝕀 (slowfib₂ n) ≡ run𝕀 (slowfib₂ (n+2) -}
  run𝕀 (slowfib₂ (n+2))
```

□

This proof fairly straightforwardly unfolds the monadic operations $\gg\!=$ and *return*, and simplifies intermediate computations with a few well-chosen auxiliary lemmas (1)–(4). These lemmas can also be proven with equational reasoning.

Some proofs are more complex than others. Take for instance a proof for the statement that the memoized and non-memoized variants of the Fibonacci function are equivalent:

Theorem 2 (NON-INTERFERENCE OF MEMOIZATION)

$$\mathit{slowfib}_2 \equiv \mathit{fastfib}_2$$

The proof of this theorem is more complex as it mutually depends on an invariant of the memo table t that we must also show to be preserved by $\mathit{fastfib}_2$:

$$\forall n.\quad \mathit{lookup}\ n\ t\ \text{`}\mathit{mplus}\text{`}\ \mathit{Just}\ (\mathit{slowfib}_2\ n) \quad\equiv\quad \mathit{Just}\ (\mathit{slowfib}_2\ n)$$

which expresses that, if the table t contains an entry for n, this entry equals $\mathit{slowfib}_2\ n$. When proofs become more intricate like this, it is useful to turn to a proof assistant, like Coq or Isabelle, that directly supports proving statements about purely functional programs. These assistants add formal rigor to the process, and bolster our confidence in the validity of the proof we write.

4 Functional Programming Tools for Modular Reasoning

The equational reasoning approach followed in the previous section has three obvious disadvantages that stem from the non-modularity of the reasoning approach:

Whole Program Knowledge The approach is a non-modular whole-program approach. The proof involves the definitions of all three parts of the program: that of the base component, the mixin and the monad.
Hence, the approach does not work in the case one of the component's implementations is not available. Moreover, if the implementations are available, it requires sufficient familiarity of the programmer who writes the proof.

Non-Trivial Activity Even though the proofs are fairly straightforward for simple situations like the logging example, they are rather longwinded and do require an effort from the programmer. Moreover, not all programers are familiar with a theorem proving tool like Coq, and, even if they are, its use does complicate the program development process.

Limited Proof Scope At the same time the gain is limited. For example, we establish that the logging mixin does not interfere with the Fibonacci component, but we cannot conclude anything about how it interferes with other base components. That requires additional proofs, one for each base component that we want to pair it with.

As we shall see in Section 5 it is possible to state very general harmless mixin theorems that only require the definition of the mixin. With those theorems we can avoid tiresome proofs like that of *logfib* entirely.

This section shows a number of useful reasoning tools that make such general harmless mixins theorems possible:

- **Effectful Reasoning:** we can avoid having to know the definitions of monadic operations by looking only at the algebraic properties of these operations.
- **Parametricity Properties of Effectful Components:** we can know which effects a component uses just by looking at the type of the component.
- **Interference Combinators:** we can enforce various control and data flow patterns using combinators.

The remainder of this section discusses each of these in more detail.

4.1 Effectful Reasoning

Effectful reasoning relies only on the algebraic properties of effectful operations to reason about programs. This avoids the use of concrete monad definitions for monadic operations like $\gg\!=$, return, *get* or *put*; and breaking the abstraction provided by the general monad interface. Moreover, effectful reasoning has the advantage that it is possible to reason about polymorphically typed programs, where the monad type is only constrained and not instantiated to a monomorphic type. For example, consider the program:

$$tick = get \gg\!= put \circ (+1)$$

The most general type for *tick* (and the type that Haskell infers) is:

$$tick :: \mathbb{S}_{\mathrm{M}}\ Int\ m \Rightarrow m\ ()$$

This type nicely abstracts from any concrete state monad implementation. All that we need to know to type-check this program is that whatever instantiation of *m* we pick, this instantiation supports the stateful operations *get* and *put* (which are members of the $\mathbb{S}_{\mathrm{M}}$ class). Possible instantiations of *m* are, for example:

$$\begin{array}{l}\textbf{type}\ M_1 = \mathbb{S}\ Int\ ()\\ \textbf{type}\ M_2 = \mathbb{S}_{\mathrm{T}}\ Int\ (\mathbb{R}\ String)\ ()\\ \textbf{type}\ M_3 = \mathbb{R}_{\mathrm{T}}\ String\ (\mathbb{S}\ Int)\ ()\end{array}$$

We want to reason about *tick* in a way that is valid for all possible instantiations of *m*. This is ultimately crucial for dealing with polymorphic monadic components as the ones discussed in this paper. For this purpose we already have presented algebraic properties for the operations $\gg\!=$ and *return*, known as the monad laws, in Section 2.2; these are valid for any instantiation of the monad *m*. What is still missing are the algebraic properties for the state-specific monadic operations *get* and *put*.

Algebraic Properties for Stateful Effects Five laws govern the semantics of the *get* and *put* methods:

Definition 2 (State Laws)

$$\begin{array}{rcll} get \gg m & \equiv & m & (\textsc{Get-Query})\\ get \gg\!= \lambda s \rightarrow get \gg\!= f\ s & \equiv & get \gg\!= \lambda s \rightarrow f\ s\ s & (\textsc{Get-Get})\\ put\ x \gg put\ y & \equiv & put\ y & (\textsc{Put-Put})\\ put\ x \gg get & \equiv & put\ x \gg return\ x & (\textsc{Put-Get})\\ get \gg\!= put & \equiv & return\ () & (\textsc{Get-Put}) \end{array}$$

Informally, the (Get-Query) law expresses that a *get* whose result is not used has no impact at all. The (Get-Get) law states that successive *get* operations return the same value, while (Put-Put) captures that successive *put* operations overwrite one another. Finally, (Put-Get) says that *get* returns the value just written by *put*, and (Get-Put) states that writing the value just read has no impact.

These five laws state the properties that each implementation of $\mathbb{S}_{\mathrm{M}}$ should conform to. Provided with these five laws and the three monads laws, it becomes possible to reason about polymorphic monadic programs like *tick*.

We illustrate this by following up on an example of Gibbons and Hinze (2011), who show, for all implementations of *tick* where *m* is any monad (not necessarily a state monad) and only using the monad laws, that

Theorem 3 (HANOI TICKS)

$$hanoi\ n \equiv rep\ (2*n-1)\ tick$$

where *hanoi* and *rep* are defined as:

$$\begin{aligned} &hanoi\ 0 &&= return\ ()\\ &hanoi\ (n+1) &&= hanoi\ n \gg tick \gg hanoi\ n\\ &rep\ 0\ \ \ \ \ \ \ \ \ mx &&= return\ ()\\ &rep\ (n+1)\ mx &&= mx \gg rep\ n\ mx \end{aligned}$$

In words, the theorem states that *hanoi n* is equivalent to $2*n-1$ successive *tick*s.

By exploiting the monad state laws above, and the details of our *tick* implementation, but not the particular monad state implementation, we can show a more interesting equation, that actually allows us to optimize the program.

Theorem 4 (TICK FUSION)

$$rep\ n\ tick \equiv get \ggg put \circ (+n)$$

In words, this theorem expresses that *n* successive *tick*s are equivalent to adding in a single step *n* to the state.

Proof
The proof proceeds by induction on *n*. For $n \equiv 0$ we have,

$$\begin{aligned} &\ \ rep\ 0\ (get \ggg put \circ (+1))\\ \equiv&\ \ \{\text{-unfold rep 0 -}\}\\ &\ \ return\ ()\\ \equiv&\ \ \{\text{-GET-PUT law -}\}\\ &\ \ get \ggg put\\ \equiv&\ \ \{\text{-id is neutral element of function composition -}\}\\ &\ \ get \ggg put \circ id\\ \equiv&\ \ \{\text{-0 is neutral element of addition -}\}\\ &\ \ get \ggg put \circ (+0) \end{aligned}$$

and for $n+1$, we have:

$$\begin{aligned} &\ \ rep\ (n+1)\ (get \ggg put \circ (+1))\\ \equiv&\ \ \{\text{-unfold rep -}\} \end{aligned}$$

$$
\begin{aligned}
& \quad \mathit{get} \gg\!= \mathit{put} \circ (+1) \gg \mathit{rep}\ n\ (\mathit{get} \gg\!= \mathit{put} \circ (+1)) \\
\equiv & \ \{\text{-induction hypothesis -}\} \\
& \quad \mathit{get} \gg\!= \lambda x \to \mathit{put}\ (x+1) \gg \mathit{get} \gg\!= \lambda y \to \mathit{put}\ (y+n) \\
\equiv & \ \{\text{-}\textsc{Put-Get}\text{ law -}\} \\
& \quad \mathit{get} \gg\!= \lambda x \to \mathit{put}\ (x+1) \gg \mathit{return}\ (x+1) \gg\!= \lambda y \to \mathit{put}\ (y+n) \\
\equiv & \ \{\text{-}\textsc{Return-Bind}\text{ law -}\} \\
& \quad \mathit{get} \gg\!= \lambda x \to \mathit{put}\ (x+1) \gg \mathit{put}\ (x+1+n) \\
\equiv & \ \{\text{-}\textsc{Put-Put}\text{ law -}\} \\
& \quad \mathit{get} \gg\!= \lambda x \to \mathit{put}\ (x+1+n) \\
\equiv & \ \{\text{-associativity and commutativity of + and fold (.) -}\} \\
& \quad \mathit{get} \gg\!= \mathit{put} \circ (n+1)
\end{aligned}
$$

□

Algebraic Properties for Other Types of Effects Other types of effects like exceptions, non-determinism, or the reader and writer monads also have similar laws that can be exploited when reasoning about effects. Liang et al. (1996) showed laws for the reader monad. More recently, since the publication of the conference version of our paper (Oliveira *et al.*, 2010) (where 4 of the above 5 algebraic properties for stateful effects were introduced), Gibbons and Hinze (2011) have explored and significantly extended the framework of algebraic properties for monadic effects. We refer to their work for the laws about other types of effects.

4.2 Type-based Reasoning: Parametricity

By making effects explicit in the types, we can learn a lot about possible effect interactions by just looking at the types. For example, just by analyzing types it is possible to discover that a component is pure (that is it does not use any side-effects) or that it is an impure component that uses some specific type of effects.

Pure Components We say that a monadic component with a type of the form

$$p :: \mathit{Monad}\ m \Rightarrow \mathit{Open}\ (a \to m\ b)$$

is *pure* because no effects can be produced by a component of this type. Voigtländer (2009) explains that this follows from the type in a language with strong parametricity properties like Haskell. The explicit effect m is a type variable only constrained to be a monad and, consequently it cannot produce any effects of its own, because it is unaware of the particular effects used. Although mixin purity comes essentially for free in Haskell, in other languages it is much harder to enforce, and it often requires sophisticated program analysis (Salcianu & Rinard, 2005). While purity imposes severe limitations on mixin code, it is also easiest to see that this code will not interact through effects at all.

Following Voigtländer's parametricity approach (see Appendix A for a summary), we can derive free formal theorems from pure components. One such theorem is that:

Theorem 5 (PURE COMPONENT)

$$new\ p \quad \equiv \quad return \circ run\mathbb{I} \circ new\ p$$

which is a formal way of saying that p does not produce any effects itself. The proof follows directly from Voigtländer's proof (Voigtländer, 2009).

An example of a pure monadic component is fib_2 (Figure 3).

Impure Components We say that a monadic component with a type of the form

$p :: \mathbb{S}_{M}\ s\ m \Rightarrow Open\ (a \to m\ b)$

is *impure* because the monad m allows a particular kind of effects to be used in the component p. In this case, p is a computation that (potentially) reads and writes a state of type s, and consequently can perform some effects.We should remark that, although for this particular example we used state, the assumption of any other kind of effect (like *exceptions*, *non-determinism* or *continuations*) would make the mixin equally impure for similar reasons.

Based on parametricity, we can derive free theorems for impure components too. For instance,

Theorem 6 (STATEFUL COMPONENT)

$$new\ p\ x \quad \equiv \quad \begin{array}{l} \textbf{do}\ s_0 \leftarrow get \\ \quad \textbf{let}\ (r, s_1) = run\mathbb{S}\ (new\ p\ x)\ s_0 \\ \quad put\ s_1 \\ \quad return\ r \end{array}$$

expresses the intuition that a stateful component *new p* can be summarized as a purely functional core that computes a state change and a single *get-put* sequence on the outside to effect the update. Appendix B lists the parametricity-based proof of this theorem.

Examples of impure monadic components are *log* and *memo* mixins, or the evaluator presented in Figure 5.

4.3 Interference Combinators

MRI provides *interference combinators* to *enforce* different interference patterns at component composition time. These interference combinators use type-based reasoning and associate 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.

Our interference combinators are inspired by Rinard et al.'s (2004) classification system for interference patterns that can occur between AOP advice and advised programs. Their classification system partitions interference forms in two major types: *direct interference* indicates the presence of control flow manipulations, whereas *indirect interference* indicates the presence of data flow manipulations.

Note that in order to draw formal conclusions from the use of these interference combinators, they must be combined with the other reasoning techniques presented in this section. Section 5 will do so and make strong formal statements.

4.3.1 Enforcing Control Flow Properties

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

Combination: A mixin can call *super* any number of times.
Replacement: There are no calls to *super* in mixins.
Augmentation: A mixin that calls *super* exactly once and does not modify the arguments to *super* or the value returned by *super*.
Narrowing: A mixin that calls *super* at most once and does not modify the arguments to *super* or the value returned by *super*.

We discuss next the combinators that capture the above interference patterns for mixins.

Combination The existing $\oplus$ combinator does not enforce any interference properties. The $\oplus$ operator already composes a mixin of the general form *Open s* with the base component.

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

$$\begin{aligned}&\textbf{type}\ \mathit{Replace}\ s = s\\&\mathit{replace} :: \mathit{Replace}\ s \to \mathit{Open}\ s\\&\mathit{replace}\ \mathit{rmxn} = \lambda \mathit{super} \to \mathit{rmxn}\end{aligned}$$

A replacement mixin has type *Replace s*, which is the type of a closed component. This reflects the fact that the replacement mixin is a proper base component by itself. In other words the base component's behavior is replaced (or overridden) entirely, which has the effect of destroying the usual control flow of the base component.

Augmentation The informal requirement for an augmentation mixin is that *super* is called exactly once and with the same argument as the mixin. This behavior is enforced with the *augment* combinator

$$\begin{aligned}&\textbf{type}\ \mathit{Augment}\ a\ b\ c\ m = (a \to m\ c, a \to b \to c \to m\ ())\\&\mathit{augment} :: \mathit{Monad}\ m \Rightarrow \mathit{Augment}\ a\ b\ c\ m \to \mathit{Open}\ (a \to m\ b)\\&\mathit{augment}\ (\mathit{bef}, \mathit{aft})\ \mathit{super}\ a =\\&\quad \textbf{do}\ \{c \leftarrow \mathit{bef}\ a; b \leftarrow \mathit{super}\ a; \mathit{aft}\ a\ b\ c; \mathit{return}\ b\}\end{aligned}$$

This combinator is responsible for calling *super* itself, rather than delegating this responsibility to the mixin. The augmentation mixin has type *Augment a b c m*, and it consists of two components: the first component is called *before super* 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 *super*. Moreover, the *before* part can communicate an auxiliary value *c* to the *after* part. For instance, log_1 is a logging mixin

```
log₁ :: (𝕎_M 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* for mixins:

```
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 mixin can be written with *before*:

```
dump₁ :: (𝕊_M s m, 𝕎_M 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 mixin calls *super* at most once. Hence, a runtime choice can be made between replacement or augmentation:

```
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) super x =
  do b ← p x
     if b then replace rep super x
          else augment aug super x
```

The runtime choice is made by the predicate of type $a \to m\ Bool$, based on the input of type *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 :: (𝕊_M (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 ()
```

$$
\begin{aligned}
&\mathit{aft}\ x\ r\ \_ = \textbf{do}\ \{m \leftarrow \mathit{get}; \mathit{put}\ (\mathit{insert}\ x\ r\ m)\} \\
&\mathit{rep}\ x \quad\ = \textbf{do}\ \{m \leftarrow \mathit{get}; \mathit{return}\ (m\ !\ x)\}
\end{aligned}
$$

This variant of *memo* makes it clear that *super* is called at most once.

4.3.2 Enforcing Data Flow Properties

Indirect interference is related to data flow through the possible interaction of shared effects (or data) between mixin and base component. 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 component), specific to state. Similar forms of interference occur with mixins:

Orthogonal: The mixin and method access disjoint fields. In this case we say that the scopes are orthogonal.
Independent: Neither the mixin 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 mixin may read one or more fields that the method may write but they are otherwise independent. In this case we say that the mixin scope observes the method scope.
Actuation: The mixin 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 mixin and method may write the same field. In this case we say that the two scopes interfere.

MRI 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 components, a mixin and a base component. MRI builds interference combinators from primitive combinators for individual components. These primitives express whether the mixin with effect *t* knows the type of effect *m* of the base component. 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 the plain mixin form:

$$
\begin{aligned}
&\textbf{type}\ \mathit{NIMixin}\ a\ b\ t = \forall m.(\mathit{Monad}\ m, \mathit{Monad}\ (t\ m)) \Rightarrow \mathit{Open}\ (a \to t\ m\ b) \\
&\mathit{nimixin} :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \Rightarrow \mathit{NIMixin}\ a\ b\ t \to \mathit{Open}\ (a \to t\ m\ b) \\
&\mathit{nimixin}\ \mathit{mix} = \mathit{mix}
\end{aligned}
$$

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

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

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

The types *NIMixin* and *NIBase* allow us to separate the effects that can be manipulated by the mixin from the effects that can be manipulated by the base component. The type system guarantees that this is indeed the case.

The type signatures of the mixin log_1 and base component *beval* are not adequate to establish non-interference. In fact, it is possible to obtain both non-interference and interference, depending on the instantiation of the monad. The non-interference combinators confront us with this issue: both *nimixin* (*augment* (log_1 `"eval"`)) and *nibase beval* are ill-typed.

Recall that the type signature of log_1 `"eval"` is

$$log_1\ \texttt{"eval"} :: (\mathbb{W}_{\mathrm{M}}\ String\ m, Show\ a, Show\ b) \Rightarrow Open\ (a \to m\ b)$$

while *nimixin* expects the type

$$\forall m'.(Monad\ m', Monad\ (t\ m')) \Rightarrow Open\ (a \to t\ m'\ b)$$

The problem is that the former type does not cleanly split the monad *m* into two parts: the transformer *t* which is used by the log_1 mixin, and the rest m' underneath which is exclusively at the disposal of the base component. To respect the non-interference pattern and obtain a well-typed instance, we split the type *m* by instantiating it to $\mathbb{W}_{\mathrm{T}}$ *String* m'. This happens when we supply the following signature:

$$\begin{array}{l} log_2 :: (Show\ a, Show\ b) \Rightarrow NIMixin\ a\ b\ (\mathbb{W}_{\mathrm{T}}\ String) \\ log_2 = augment\ (log_1\ \texttt{"eval"}) \end{array}$$

With this signature all the *tell* operations in log_1 are handled by $\mathbb{W}_{\mathrm{T}}$ and the underlying monad is not accessed.

The same problem arises in *nibase beval*, where *beval* has type

$$beval :: \mathbb{S}_{\mathrm{M}}\ Env\ m \Rightarrow Open\ (Expr \to m\ Int)$$

and *nibase* expects

$$\forall t.(MonadTrans\ t, Monad\ (t\ m')) \Rightarrow Open\ (a \to t\ m'\ b)$$

Unfortunately, for technical reasons it is not easy to split the monad for a base component. Instantiating *m* to *t* ($\mathbb{S}$ *Env*) does not mean that the *get* and *put* operations in *beval* are necessarily resolved against the embedded monad $\mathbb{S}$ *Env*. Whether or not this is the case still depends on the choice of *t* and hence is ambiguous if we leave *t* undetermined. This ambiguity causes the following code to be ill-typed:

$$\begin{array}{l} beval_1 :: NIBase\ Expr\ Int\ (\mathbb{S}\ Env) \\ beval_1 = beval \end{array}$$

We can solve this issue by explicitly defining a variant of *beval* in which the monad is explicitly split up in two parts, a monad transformer *t* and an embedded monad m'. By means of *lift* this variant resolves its uses of *get* and *put* against the embedded monad m'.

```
beval'' :: (MonadTrans t, Monad (t m'), S_M Env m') ⇒ Open (Expr → t m' Int)
beval'' this exp = case exp of
  Var s       → do e ← lift get
                   case lookup s e of
```

```
                        Just x    → return x
                        Nothing → error msg
    Assign x r → do y ← this r
                    e ← lift get
                    lift (put ((x, y) : e))
                    return y
    ...   -- The other cases identical to those in beval
```

This definition clearly fits the *NIBase* pattern. The downside is that we had to rewrite the definition of *beval*. With the help of the *monad zipper* (Schrijvers & Oliveira, 2011) this rewriting could have been avoided, but since that approach is rather technical we won't got into details here.

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

```
(⊖)   :: (MonadTrans t, Monad m, Monad (t m))
        ⇒ NIMixin a b t → NIBase a b m → Open (a → t m b)
mix ⊖ bse = nimixin mix ⊕ nibase bse
(⧁) :: (MonadTrans t, Monad m, Monad (t m))
        ⇒ Open (a → t m b) → NIBase a b m → Open (a → t m b)
mix ⧁ bse = mix ⊕ nibase bse
(⧀)   :: (MonadTrans t, Monad m, Monad (t m))
        ⇒ NIMixin a b t → Open (a → t m b) → Open (a → t m b)
mix ⧀ bse = nimixin mix ⊕ bse
(⊗)   :: (MonadTrans t, Monad m, Monad (t m))
        ⇒ Open (a → t m b) → Open (a → t m b) → Open (a → t m b)
mix ⊗ bse = mix ⊕ bse
```

Note that, unlike Rinard's categories, these combinators are not specific for state: they are parametric in the type of effect. The combinators ⊗ and ⊖ closely correspond to Rinard's interference and orthogonal categories. The ⧁ and ⧀ combinators indicate which of the two components is aware of the other's effects, which are thus shared between the two components.

For instance, the composition $log_2 \ominus beval_1$ expresses that the logging mixin 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. MRI distinguishes between such forms of interaction by imposing appropriate constraints on the monad type variable *m*.

For this purpose MRI refines $\mathbb{S}_{\mathrm{M}}$ to cater for different views:

```
class Monad m ⇒ MGet s m | m → s where
  get :: m s
class Monad m ⇒ MPut s m | m → s where
  put :: s → m ()
```

$\textbf{class}\ (\mathit{MGet}\ s\ m, \mathit{MPut}\ s\ m) \Rightarrow \mathbb{S}_{\mathrm{M}}\ s\ m$

The constraint *MGet s m* only allows reading the state *s* of monad *m*, while the class *MPut* only allows writing it. The new $\mathbb{S}_{\mathrm{M}}\ s\ m$ allows both reading and writing by subclassing both *MGet* and *MPut*. The methods *get* and *put* obey the laws presented in Section 4.1.

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

$$\begin{array}{l}
\mathit{dump_2} :: (\mathit{MGet}\ s\ m, \mathbb{W}_{\mathrm{M}}\ \mathit{String}\ m, \mathit{Show}\ s) \Rightarrow a \to m\ ()\\
\mathit{dump_2}\ \_ = \textbf{do}\ \{s \leftarrow \mathit{get}; \mathit{tell}\ (\mathit{show}\ s +\!\!+ \texttt{"\textbackslash n"})\}
\end{array}$$

With the two new constraints, MRI also defines relaxed versions of *NIMixin*:

$$\begin{array}{l}
\textbf{type}\ \mathit{ROMixin}\ a\ b\ t\ s\ = \forall m.(\mathit{MGet}\ s\ m, \mathit{Monad}\ (t\ m)) \Rightarrow \mathit{Open}\ (a \to t\ m\ b)\\
\textbf{type}\ \mathit{WOMixin}\ a\ b\ t\ s = \forall m.(\mathit{MPut}\ s\ m, \mathit{Monad}\ (t\ m)) \Rightarrow \mathit{Open}\ (a \to t\ m\ b)
\end{array}$$

Each of these forms of mixin assumes one operation on the underlying monad *m*, *get* for *ROMixin* and *put* for *WOMixin*, and both obviously assume that *t m* is a monad.

The *dump3* mixin instantiates *dump2* as a *ROMixin*:

$$\begin{array}{l}
\mathit{dump_3} :: \mathit{Show}\ s \Rightarrow \mathit{ROMixin}\ a\ b\ (\mathbb{W}_{\mathrm{T}}\ \mathit{String})\ s\\
\mathit{dump_3} = \mathit{before}\ \mathit{dump_2}
\end{array}$$

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

$$\begin{array}{l}
\mathit{observation} :: (\mathit{MGet}\ s\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \Rightarrow\\
\quad \mathit{ROMixin}\ a\ b\ t\ s \to \mathit{NIBase}\ a\ b\ m \to \mathit{Open}\ (a \to t\ m\ b)\\
\mathit{observation}\ \mathit{mix}\ \mathit{bse} = \mathit{mix} \oplus \mathit{bse}\\
\mathit{actuation} :: (\mathit{MPut}\ s\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \Rightarrow\\
\quad \mathit{WOMixin}\ a\ b\ t\ s \to \mathit{NIBase}\ a\ b\ m \to \mathit{Open}\ (a \to t\ m\ b)\\
\mathit{actuation}\ \mathit{mix}\ \mathit{bse} = \mathit{mix} \oplus \mathit{bse}
\end{array}$$

MRI puts similar constraints on the base component and distinguishes nine different forms of interference. The following table connects these nine forms to the corresponding four terms used by Rinard et al.:

	Base Component		
	MGet	*MPut*	$\mathbb{S}_{\mathrm{M}}$
Mixin			
MGet	Independent	Observation	Observation
MPut	Actuation	Interference	Interference
$\mathbb{S}_{\mathrm{M}}$	Actuation	Interference	Interference

By distinguishing between $\mathbb{S}_{\mathrm{M}}$ and *MPut*, MRI has a more fine-grained classification. $\mathit{MPut} \times \mathit{MPut}$, for instance, is only a weak form of interference. While both components write to the same state, neither's computations are affected; only the resulting state is.

While Rinard's classification is specific for state, MRI allows similar classifications for other kinds of effects. For example, with exceptions the rights to throw and catch exceptions are separated into different monad subclasses: *MonadThrow e m* for throwing an exception *e*, *MonadCatch e m* for catching, and $\mathbb{E}_{\mathrm{M}}\ e\ m$ for both. By considering the permitted operations of the mixin and base component, the possible interference patterns between them are established.

5 Harmless Mixins: Strong Guarantees of Non-Interference

This section explains how to enforce strong guarantees of non-interference for mixins with direct and indirect non-interference combinators. These strong guarantees of non-interference are inspired by Dantas and Walker (2006) notion of *harmless advice*:

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.

5.1 Harmless Mixins

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

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

$$\begin{array}{l}(\circledast) :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \Rightarrow \\ \quad \mathit{NIAugment}\ a\ b\ c\ t \rightarrow \mathit{NIBase}\ a\ b\ m \rightarrow \mathit{Open}\ (a \rightarrow t\ m\ b) \\ \mathit{mix} \circledast \mathit{bse} = \mathit{augment}\ \mathit{mix} \ominus \mathit{bse}\end{array}$$

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

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

Theorem 7 (HARMLESS MIXIN)
Consider a base component *bse* and mixin *mix* with the types:

$$\begin{aligned}&\mathit{bse} :: \forall t.(\mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m_1)) \Rightarrow \mathit{Open}\ (a \to t\ m_1\ b)\\ &\mathit{mix} :: \forall m.(\mathit{Monad}\ m, \mathit{Monad}\ (t_1\ m)) \Rightarrow \mathit{Augment}\ a\ b\ c\ (t_1\ m)\end{aligned}$$

where a, b, c, m_1 and t_1 are arbitrary given types with m_1 a monad and t_1 a monad transformer. Then mixin *mix* is harmless with respect to *bse*:

$$\pi \circ (\mathit{new}\ (\mathit{mix} \circledast \mathit{bse})) \equiv \mathit{run}\mathbb{I}_{\mathrm{T}} \circ (\mathit{new}\ \mathit{bse})$$

for any projection function $\pi :: \forall m\ a.\mathit{Monad}\ m \Rightarrow t_1\ m\ a \to m\ a$ that satisfies the law:

$$\pi \circ \mathit{lift} \equiv \mathit{id} \qquad (\text{PROJECT-LIFT})$$

Informally, the theorem states that, if we ignore the effects introduced by the mixin, the advised program is equivalent to the unadvised program. The role of the projection function π is to ignore the effects introduced by the mixin. The PROJECT-LIFT law expresses the intuition that projection has no impact if there are no effects.

Proof The proof essentially combines the three important reasoning principles: 1) equational reasoning over the combinator definitions, 2) algebraic reasoning with the monad laws and the PROJECT-LIFT law, and 3) parametricity of the mixin and base component types. Here we sketch the three high-level steps of the proof; Appendix C provides the full details:

1. First, we show how to convert between the self-explanatory form of augmentation mixin we have used so far 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 mix = λsuper →
    λa → mix a >>= λaft →
      super a >>= λr →
      aft r     >>= \_ →
      return r
```

Lemma 1
Consider augmentation mixin $(bef, aft) :: (a \to t_1\ m_1\ c, a \to b \to c \to t_1\ m_1\ ())$, then we have that:

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

where m_1 is a *Monad* and t_1 is a *MonadTrans*.

The proof of this lemma is based on equational reasoning and the monad laws.

2. Then, exploiting parametricity, we derive two free theorems, one for the *around* mixin:[3]

Lemma 2
Consider a function $f :: \forall m. Monad\ m \Rightarrow a \to m\ (b \to m\ c)$, then we have that:

$$f_{m_1} \equiv (out\ (return \circ out\ (return \circ run\mathbb{I}) \circ run\mathbb{I}))\ f_{\mathbb{I}}$$

and one for the base component:

Lemma 3
Consider a function $f :: \forall t. MonadTrans\ t \Rightarrow (a \to t\ m_1\ (b \to t\ m_1\ ())) \to a \to t\ m_1\ b$ with m_1 an arbitrary monad, then we have that:

$$out\ \pi \circ f \equiv$$
$$out\ run\mathbb{I}_T \circ f \circ out\ (\mathbb{I}_T \circ fmap\ (out\ (\mathbb{I}_T \circ \pi)) \circ \pi)$$

for any $\pi :: \forall m, a. Monad\ m \Rightarrow t_1\ m\ a \to m\ a$, with t_1 an arbitrary monad transformer, that satisfies the PROJECT-LIFT law.

While parametricity is the core technique for proving these two theorems, the necessary logical relations are established by means of equational reasoning, the PROJECT-LIFT law and the monad laws.
Note that 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 mixin may introduce non-termination and partiality. This is our counterpart of “may change the termination behavior” in Dantas’s and Walker’s definition.

3. Finally, we prove the main theorem in a big equational reasoning proof. This proof relates the left-hand side to the right-hand side of the theorem’s equality in a number of successive equality-preserving steps. These steps involve folding and unfolding combinator definitions, the three lemmas above, the monad laws, the PROJECT-LIFT law, and simple β- and η reductions.

[3] Here, $out = (\circ)$ applies a function to the output of another function.

5.1.1 Harmless Effects

In order to suit the Harmless Mixin theorem, the mixin 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.

Writer For the $\mathbb{W}_\mathrm{T}$ monad transformer we define the following projection function:

$$\pi_W :: \forall w\ m\ a.(\mathit{Monad}\ m, \mathit{Monoid}\ w) \Rightarrow \mathbb{W}_\mathrm{T}\ w\ m\ a \to m\ a$$
$$\pi_W\ m = \mathit{run}\mathbb{W}_\mathrm{T}\ m \gg\!= \mathit{return} \circ \mathit{fst}$$

It is indeed suitable:

Lemma 4
The function π_W is a suitable function for the Harmless Mixin theorem:

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

See Appendix D.1 for the proof.

With the help of π_W, the Harmless Mixin theorem establishes that the logging mixin is harmless:

$$\pi_W \circ \mathit{new}\ (\mathit{log}_2\ \texttt{"eval"} \circledast \mathit{beval}_1) \equiv \mathit{run}\mathbb{I}_\mathrm{T} \circ \mathit{new}\ \mathit{beval}_1$$

State We can also define a suitable projection function for the $\mathbb{S}_\mathrm{T}$ monad transformer:

$$\pi_S :: \forall s\ m\ a.\mathit{Monad}\ m \Rightarrow s \to \mathbb{S}_\mathrm{T}\ s\ m\ a \to m\ a$$
$$\pi_S\ s_0\ m = \mathit{run}\mathbb{S}_\mathrm{T}\ m\ s_0 \gg\!= \mathit{return} \circ \mathit{fst}$$

Indeed, the required property holds:

Lemma 5
The function $\pi_S\ s_0$ is a suitable function for the Harmless Mixin theorem:

$$\pi_S\ s_0 \circ \mathit{lift} \equiv \mathit{id}$$

for any s_0.

See Appendix D.2 for the proof.

Other Harmless Effects There are several other harmless effects, such as $\mathbb{I}_\mathrm{T}$ with trivial projection function $\mathit{run}\mathbb{I}_\mathrm{T}$, $\mathbb{R}_\mathrm{T}$ and variations on these.

5.1.2 Harmful effects

An interesting aspect of our theorem is that harmless mixins may not introduce arbitrary effects. Only those effects for which a suitable projection function π exists, may be used in harmless mixins. Some types of effects can be harmful.

Error Consider again the $\mathbb{E}_\mathrm{T}\ e$ monad transformer of Figure 2. We can only partially define the projection function:

$$
\begin{array}{l}
\pi_E :: \forall e\ m\ a. Monad\ m \Rightarrow \mathbb{E}_\mathrm{T}\ e\ m\ a \rightarrow m\ a \\
\pi_E\ m = run\mathbb{E}_\mathrm{T}\ m \gg\!= \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
\qquad\quad Left\ e \rightarrow ??? \\
\qquad\quad Right\ x \rightarrow return\ x
\end{array}
$$

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

$$
\begin{array}{l}
\pi_E :: \forall e\ m\ a. Monad\ m \Rightarrow a \rightarrow \mathbb{E}_\mathrm{T}\ e\ m\ a \rightarrow m\ a \\
\pi_E\ d\ m = run\mathbb{E}_\mathrm{T}\ m \gg\!= \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
\quad Left\ e \rightarrow return\ d \\
\quad Right\ x \rightarrow return\ x
\end{array}
$$

but now $\pi_E\ d :: \forall e\ m. Monad\ m \Rightarrow \mathbb{E}_\mathrm{T}\ e\ m\ a \rightarrow m\ a$ fixes the type parameter a to the type of d, which is inappropriate.

Intuitively, the reason why errors are not a harmless effect is because they can change the normal control flow of a program if an error (exception) occurs.

I/O 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 component. 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 $\mathbb{W}_\mathrm{T}$, $\mathbb{R}_\mathrm{T}$ and $\mathbb{S}_\mathrm{T}$ are possible.

5.2 Harmless Observation Mixins

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

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

Now we can adapt the theorem accordingly:

Theorem 8 (HARMLESS OBSERVATION MIXIN)
Consider a base component *bse* and mixin *mix* with the types:

$$\begin{aligned}&bse :: \forall t.(MonadTrans\ t, Monad\ (t\ m_1)) \Rightarrow Open\ (a \to t\ m_1\ b)\\ &mix :: \forall m.(MGet\ s\ m, Monad\ (t_1\ m)) \Rightarrow Augment\ a\ b\ c\ (t_1\ m)\end{aligned}$$

where a, b, c, s, m_1 and t_1 are arbitrary given types with m_1 a $\mathbb{S}_{\mathrm{M}}$ s and t_1 a monad transformer. Then the mixin *mix* is harmless with respect to *bse*:

$$\pi \circ (new\ (mix \odot bse)) \equiv run\mathbb{I}_{\mathrm{T}} \circ (new\ bse)$$

for any projection function $\pi :: \forall m\ a.Monad\ m \Rightarrow t_1\ m\ a \to m\ a$ that satisfies the PROJECT-LIFT law.

Proof The proof is similar in style to that of the Harmless Mixin theorem. The main difference lies in the fact that the mixin knows more about the *m* type parameter. As a consequence, weaker parametricity results are obtained. The core insight is that we can make up for this loss of parametricity by exploiting the GET-QUERY and GET-GET laws. We refer to Appendix E for the proof details.

Example Theorem 8 establishes that the dumping mixin is harmless:

$$\pi_W \circ new\ (dump_3 \odot beval_1) \equiv run\mathbb{I}_{\mathrm{T}} \circ new\ beval_1$$

6 Related Work

6.1 Reasoning in Functional Programming

Our work shows how functional programming reasoning techniques can be applied to a notoriously hard problem: modularly reasoning about inheritance in the presence of side-effects. To address this challenging problem we have had to cast it in the right combinator-based formulation in which we could bring the synergy of parametricity, equational reasoning and algebraic laws to bear. We next discuss in more detail related work on these reasoning techniques for purely functional programs.

Equational Reasoning There is a long tradition of reasoning about purely functional programs. A great benefit of purity is that it allows *equational reasoning*: that is reasoning about programs using simple algebraic equations (much like high-school algebra). The seminal book on “The Algebra of Programming” (Bird & De Moor, 1997) is a highlight of this reasoning approach.

Non-modular monadic reasoning However, only recently there has been some interest on exploring such reasoning techniques for monadic programs. Although monads (Wadler, 1992b) are a purely functional way to encapsulate computational-effects, programs using monads are challenging to reason about. The main issue is that monads provide an abstraction over purely functional models of effects, allowing functional programmers to write programs in terms of abstract operations like $\gg\!=$, *return*, or *get* and *put*. One way to reason about monadic programs is to remove the abstraction provided by such operations (Hutton & Fulger, 2008). We follow this approach in our reasoning technique presented in Section 3. However, as discussed in more detail in Section 4, there are several drawbacks to this approach. Most importantly this approach is fundamentally non-modular.

Modular monadic reasoning Using *parametricity* (Reynolds, 1983; Wadler, 1989) and algebraic laws about effectful operations it is possible to modularly reason about monadic programs. Voigtländer (2009) has shown how to derive parametricity theorems for type constructor classes such as *Monad*. This technique plays a crucial role in our modular reasoning approach, as it allows us to derive theorems about effectful programs without knowing the concrete effects used. However, parametricity alone is not enough to establish theorems such as the *Harmless Observation Mixin* theorem (see Section 5.2), which allows mixins to read the state of the base component. To account for this theorem we need algebraic laws about stateful effects (see Section 4.1). Liang et al. (1996) presented laws for reader monads. In the conference version of this paper (Oliveira *et al.*, 2010) we presented 4 laws about state. Since then, Gibbons and Hinze (2011) presented an additional law for state (the GET-PUT law) and have significantly explored algebraic laws for many other types of effects.

Non-modular monadic reasoning about FOP In the context of Feature-Oriented Programming (FOP) Prehofer (1999) defines a notion similar to Harmless Mixin, 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 mixins 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 Mixin. Exploiting this syntactic pattern enables reasoning by induction on operation sequences and equational reasoning to prove a Harmless Mixin-like theorem. Prehofer's reasoning approach is closest to the reasoning techniques presented in Section 3, in the sense that it is a non-modular approach (requires all the definitions), and it can only be used to reason about individual compositions of mixin and base component. Like our non-modular reasoning techniques, his approach supports reasoning about harmlessness that is subject to preconditions and invariants. We believe that given sufficiently polymorphic mixins it should be possible to use parametricity in Prehofer's setting to prove that a mixin is conservative regardless of the base component, thus allowing for more modular reasoning techniques similar to the ones in Section 4.

Prehofer (2006) 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 mixins. Using our approach, the uniformity of composition seems to suggest that the composition of two harmless mixins is always harmless, but this needs further investigation.

6.2 Monads and Modularity

Monad Transformers and Modular Interpreters Liang et al. (1995) proposed *Monad Transformers and Modular Interpreters* (MTMI) to show the benefits of monads and monad transformers for modularity purposes. With this technique, *modular development* of components is possible. While our approach is similar in several ways, there are two important differences. The first difference is that Liang et al. focus on *modularization of an interpreter's basic behavior according to the different language features*, while we consider the *modularization of orthogonal concerns on top of base programs*.

The second, and more fundamental, difference between the two works concerns *reasoning* and *abstraction* (or *encapsulation*) properties. We discuss these in more detail next:

- *Modular Reasoning* - Our technique supports modular reasoning and, in particular, it supports *separate compilation*. In MTMI, separate compilation *is not* supported. In their approach there are three types (*Value*, *Term* and *InterpM*) whose definitions need to be changed whenever a new component is added. However, these types are used by *all* modular components and, consequently, a change in one of these types implies recompilation of all components. This renders modular reasoning about individual components impossible since the static types may vary depending on particular instantiations of the components.
- *Encapsulation and Interference* - The type *InterpM* denotes the monad that is used by all the components in MTMI. The different parts of the monad are visible to all components (every component knows the type *InterpM*), which means that every components can interact with any part of the monad stack. In contrast, in our approach, parametric polymorphism ensures that a component can only access the parts of the monad stack that it is supposed to. In other words our approach offers encapsulation of effects while MTMI does not. As a consequence, MTMI cannot provide non-interference guarantees.

In conclusion, while MTMI supports (to a large extent) *modular development* of components, it does *not* support modular reasoning or reasoning about non-interference.

Modularity issues with Monads Transformers There are several modularity issues (and solutions), related to monad transformers, reported in the literature. In our work we rely on the monad transformer library (MTL), which is based on Liang et al.'s (1995) work and as such suffers from these issues.

The most relevant issue for us is that programs that use stacks of monads transformers (usually) have to impose very strong constraints on the orderings of the transformers in the stack. In general, components in a program that uses a monad stack with more than one monad layer of the same type (for example two state monad layers) have to be aware of the structure of the monad stack, in order to access the right monad layer. This issue was the motivation for Schrijvers and Oliveira's (2011) proposal for the *monad zipper* and *monad views*, which provide an alternative way to manage the monad stack without imposing a tight coupling between a monadic component and the structure of the monad stack. In that work a variant of the MTL is proposed. Our work could be readily adapted to work with that variant of the MTL and as such avoid this problem.

Jaskelioff (2008) reports various other issues related to the design of the MTL. Most pressingly, a deficiency of the MTL design is that when a new type of monad transformer is added, the interacting behavior between the new transformer and the existing transformers must be defined individually for each case. This is ad-hoc, non-modular and requires a growing number of instances each time a new monad transformer is added to the framework. Fortunately this issue is not so pressing for us because we (usually) work with existing effects (like state, IO or exceptions) and, as such, do not have to add new transformers to the framework.

Other Effect Models In this article we have focused on the predominant model of effects in purely functional programming: monads and monad transformers. However, other useful models have been proposed, such as *applicative functors* (McBride & Paterson, 2008) and *arrows* (Hughes, 1998), with their own axioms and modularity properties. It makes for interesting future work to adapt the developments of this article to those alternatives.

6.3 Modular Reasoning and Interference in AOP

In this paper we have focused on inheritance of functions, which is closely related to AOP advice. Next we discuss work on modular reasoning and interference in the context of AOP.

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 components with respect to impure advice. Reasoning about effectful base components 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.

Translucid contracts (Bagherzadeh *et al.*, 2011) are grey-box specifications which describe control-flow properties required by advice and advised code. Using structural refinement, the specifications are used to enforce the control-flow properties in implementations. As such translucid contracts allow programmers to understand interactions between advice and advised code without requiring them to know about implementations. Our control-flow interference combinators play a similar role to translucid contracts by statically enforcing control-flow patterns using the type system.

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

Dantas and Walker (2006) propose a type-and-effect system for identifying harmless advice on the MinAML core language (Ligatti *et al.*, 2006): protection domains prevent information flow from advice to base component. Their modular analysis supports a formal result similar to our Harmless Observation Mixin 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 (2002) identify that observers (harmless observation mixins) do not change the specification of the advised module. Later, Clifton et al. (2007) 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 mixin, 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. (2004) 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. However, their language is only partially defined, no equational laws for effects are provided and no theorem is stated. There are two important differences with MRI. Firstly, to reason about non-interference, they require the aspect definitions to apply their equational laws, while MRI only looks at the types of mixins. Secondly, they focus on aspect/aspect interaction and overlapping pointcuts and do not address aspect/base program interaction. While this article focuses on the mixin/base component interference, the same approach applies equally to the interaction of two mixins.

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 proved.

Katz (1993) presents a much earlier classification, for *superimpositions* in the context of distributed programming, which he later refines (Katz, 2006). He distinguishes *spectative*, *regulative invasive* superimpositions. Spectative superimpositions are akin to harmless observation mixins; regulative superimpositions affect which actions happen, but do not change the nature of the actions themselves; and invasive superimpositions can change anything.

In summary, existing approaches to non-interference formulate special-purpose program analyses or type systems. A major advantage of MRI 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.

6.4 Modular Reasoning and Interference in OOP

The problems of modular reasoning and interference are closely related and often come together hand-in-hand. Because of this, in object-oriented programming the two problems are usually not clearly distinguished. Normally, when referring to modular reasoning in OOP, what is meant is the problem of being able to reason about a class in the presence of subclassing (Stata & Guttag, 1995; Leino, 1998; Ruby & Leavens, 2000; Müller *et al.*, 2003). One problem is that it is often the case that, in order to define a new subclass, either knowledge about the implementation or some undocumented behavioral assumptions about the superclass are needed (Kiczales & Lamping, 1992; Lamping, 1993). Additionally, it is often unclear whether the essential behavior of the superclass is going to be preserved by subclasses because both data and control flow interference can be introduced.

Following some observations by Lamping (1993) on how methods in classes are related to each other, Stata and Guttag (1995) proposed to address the problems above using specifications extended with a notion of *groups*. Groups capture all methods in a class that directly manipulate some private field. With Stata and Guttag's approach a subclass that overrides one method in a group also needs to override all the other methods in the same group. Furthermore, the behavior of methods is ensured using traditional specifications that describe possible class invariants and pre and post conditions of the methods. A similar approach was proposed by Ruby and Leavens (2000). Their approach consist of an extension of JML with subclassing contracts, for which each method states the protected and public fields that it accesses and the method calls required by the method. This approach allows more flexibility when subclassing when compared to the groups proposed by Stata and Guttag, since it is sometimes possible to override a method without overriding all the methods that access the same private fields. Like with Stata and Guttag's approach ensuring that the behaviour of methods is preserved is achieved with traditional method specifications. Interestingly, specifications themselves are prone to modular reasoning problems, which leads to a related line of work aimed at providing modular specification of properties (Leino, 1998; Müller *et al.*, 2003).

The main difference between our work and the existing work on modular reasoning in OOP is that we focus on providing a generic model for inheritance that supports reasoning from inception, while most previous work tries to develop specification and reasoning principles a posteriori for existing OOP technologies. Our approach is aimed at understanding the essential problems that lead to the difficulties in reasoning in existing IP languages; and fostering the development of new programming languages in which such problems are addressed from scratch. A difficulty that we have not yet addressed in our model, but which is highly relevant for OOP, relates to control flow interference. As shown in Section 5, it is possible to have strong-guarantees of non-interference for functions, but in the case of objects matters are a bit more complicated because each method can interfere with other methods of the same object. Nevertheless, we do not think this poses a fundamental difficulty.

6.5 Functional AOP systems

There has been some interest on integrating AOP and functional programming. However, this poses quite different and new challenges compared to integrating AOP in an OO language, especially if the functional language is pure (Wang & Oliveira, 2009).

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.*, 2011); (Chen *et al.*, 2007)) and AspectML (Dantas *et al.*, 2008), and 2) *lightweight dynamically typed* approaches like 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* (De Fraine & Braem, 2007) and expressing *dynamically deployed* aspects (Tanter, 2008). In some sense, MRI 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 mixins instead of the pointcut model. In MRI, "features" (such as *first-class*, *polymorphic* and *inferable types for* mixins) come for free. In language-based approaches adding support for each of these features is non-trivial, and only AspectML supports all of them.

Chen et al. (2011) proposed a monadic semantics for AspectFun. The idea is to have a source language with private, local state for advice. Programs with local state are translated into monadic programs using a type-directed algorithm. Thus, like our work, monads model (stateful) effects, however, unlike our work (non-modular) code transformation is used instead of mixins for weaving components. Although the focus of their work is not modular reasoning, a consequence of only allowing local, private state for advice is that data-flow interference between advice and other components cannot occur. However control-flow interference can still happen. This gives them some non-interference guarantees (at the cost of expressiveness), but it is insufficient to automatically establish harmless advice.

7 Conclusion

MRI 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 is possible by looking at the interfaces only.

MRI provides a purely functional model of inheritance with effects. The benefit of being purely functional is that many powerful reasoning techniques become available. Parametricity and algebraic laws about effects are powerful forms of modular reasoning that complement basic equational reasoning. Together they allow MRI to provide effective modular reasoning techniques for non-interference of tightly coupled IP components, which is a notoriously hard problem in the literature.

Besides providing a modular reasoning framework, an additional benefit of MRI is that it provides a simple and lightweight model of inheritance as a Haskell library. Monadic mixins are a useful concept for functional programming and, in some sense, they can be viewed as a simple approach to AOP in Haskell.

Acknowledgments

We are grateful to Jonathan Aldrich, Benjamin Delaware, Marko van Dooren, Jeremy Gibbons, Simon Peyton Jones, Steven Keuchel, Shriram Krishnamurthi, Adriaan Moors, Janis Voigtländer, Meng Wang and the anonymous reviewers for their useful comments; and to Andres Löh for supporting `lhs2tex`.

Bruno Oliveira was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF) grant number R11-2008-007-01002-0, the Mid-career Researcher Program (2010-0022061) through NRF grant funded by the MEST, and by a grant from the Portugal-UT Austin CoLab program.

References

Aldrich, J. (2005). Open modules: Modular reasoning about advice. *Pages 144–168 of: ECOOP'05: 19th European Conference on Object-Oriented Programming*.

Bagherzadeh, M., Rajan, H., Leavens, G. T., & Mooney, S. (2011). Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. *Pages 141–152 of: AOSD'11: 10th International Conference on Aspect-Oriented Software Development*.

Bird, R. S., & De Moor, O. (1997). *Algebra of programming*. International Series in Computing Science, vol. 100.

Bracha, G., & Cook, W. (1990). Mixin-based inheritance. *Pages 303–311 of: Proceedings of the european conference on object-oriented programming on object-oriented programming systems, languages, and applications*. OOPSLA/ECOOP '90. New York, NY, USA: ACM.

Chen, K., Weng, S., Wang, M., Khoo, S., & Chen, C. (2007). A compilation model for aspect-oriented polymorphically typed functional languages. *Pages 34–51 of: SAS'07: 14th international symposium on static analysis*.

Chen, K., Weng, S.-C., Lin, J.-Y., Wang, M., & Khoo, S.-C. (2011). Side-effect localization for lazy, purely functional languages via aspects. *Higher-order and symbolic computation*, 1–39.

Clifton, C., & Leavens, G. T. (2002). Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. *Pages 33–44 of: FOAL'02: 1st Workshop on Foundations of Aspect-Oriented Languages*.

Clifton, C., Leavens, G. T., & Noble, J. (2007). MAO: Ownership and effects for more effective reasoning about aspects. *Pages 451–475 of: ECOOP'07: 21sth European Conference on Object-Oriented Programming*.

Cook, W., & Palsberg, J. (1989). A denotational semantics of inheritance and its correctness. *Pages 433–443 of: Conference proceedings on object-oriented programming systems, languages and applications*. OOPSLA '89.

Cook, W. R. (1989). *A denotational semantics of inheritance*. Ph.D. thesis, Brown University.

Dahl, O.-J., & Nygaard, K. (1966). Simula: An ALGOL-based simulation language. *Communications of the acm*, **9**(9), 671–678.

Dantas, D. S., & Walker, D. (2006). Harmless advice. *Pages 383–396 of: POPL'06: 33rd Symposium on Principles of Programming Languages*.

Dantas, D. S., Walker, D., Washburn, G., & Weirich, S. (2008). AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, **30**(3), 1–60.

De Fraine, B., & Braem, M. (2007). Requirements for reusable aspect deployment. *Pages 176–183 of:* Lumpe, Markus, & Vanderperren, Wim (eds), *Software composition*. Lecture Notes in Computer Science, vol. 4829. Springer Berlin / Heidelberg.

Douence, R., Fradet, P., & Südholt, M. (2004). Composition, reuse and interaction analysis of stateful aspects. *Pages 141–150 of: AOSD'04: 3rd International Conference on Aspect-Oriented Software Development*.

Dutchyn, C., Tucker, D. B., & Krishnamurthi, S. (2006). Semantics and scoping of aspects in higher-order languages. *Science of computer programming*, **63**(3), 207–239.

Flatt, M., Krishnamurthi, S., & Felleisen, M. (1998). Classes and mixins. *Pages 171–183 of: Popl*.

Gibbons, J., & Hinze, R. (2011). Just do it: Simple monadic equational reasoning. *Pages 2–14 of: ICFP'11: 16th International Conference on Functional Programming*.

Hughes, J. (1998). Generalising monads to arrows. *Science of computer programming*, **37**, 67–111.

Hutton, G., & Fulger, D. (2008). Reasoning About Effects: Seeing the Wood Through the Trees. *Proceedings of the symposium on trends in functional programming*.

Jaskelioff, Mauro. (2008). Monatron: An Extensible Monad Transformer Library. *Pages 233–248 of: IFL'08: 20th international conference on Implementation and application of functional languages*.

Jones, Mark P. (2000). Type classes with functional dependencies. *Pages 230–244 of: Esop'00*.

Katz, S. (1993). A superimposition control construct for distributed systems. *Acm trans. program. lang. syst.*, **15**(2), 337–356.

Katz, S. (2006). Aspect categories and classes of temporal properties. *Transactions on aspect-oriented software development I*, **3880**, 106–134.

Kiczales, G., & Lamping, J. (1992). Issues in the design and specification of class libraries. *Pages 435–451 of: OOPSLA'92: 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*.

Kiczales, G., & Mezini, M. (2005). Aspect-oriented programming and modular reasoning. *Pages 49–58 of: ICSE'05: 27th International Conference on Software Engineering*.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J., & Irwin, J. (1997). Aspect-oriented programming. *Pages 220–242 of: ECOOP'97: 17th European Conference on Object-Oriented Programming*.

Lamping, J. (1993). Typing the specialization interface. *Pages 201–214 of: OOPSLA'93: 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*.

Leino, K. Rustan M. (1998). Data groups: specifying the modification of extended state. *Pages 144–153 of: OOPSLA'98: 13th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*.

Lewis, J. R., Launchbury, J., Meijer, E., & Shields, M. B. (2000). Implicit parameters: dynamic scoping with static types. *Pages 108–118 of: POPL'00: 27th Symposium on Principles of Programming Languages*.

Liang, S., & Hudak, P. (1996). Modular denotational semantics for compiler construction. *Pages 219–234 of: ESOP'96: European Symposium on Programming*. Springer-Verlag.

Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. *Pages 333–343 of: POPL'95: 22nd Symposium on Principles of Programming Languages*.

Ligatti, J., Walker, D., & Zdancewic, S. (2006). A type-theoretic interpretation of pointcuts and advice. *Science of computer programming*, **63**(3), 240–266.

Lopez-Herrejon, R., Batory, D., & Lengauer, C. (2006). A disciplined approach to aspect composition. *Pages 68–77 of: PEPM'06: Symposium on Partial evaluation and semantics-based program manipulation*.

Masuhara, H., Tatsuzawa, H., & Yonezawa, A. (2005). Aspectual Caml: an aspect-oriented functional language. *Pages 320–330 of: ICFP'05: 10th International Conference on Functional Programming*.

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

Müller, P., Poetzsch-Heffter, A., & Leavens, G. T. (2003). Modular specification of frame properties in jml. *Concurrency and computation: Practice and experience*, **15**(2), 117–154.

Oliveira, B. C. d. S., Schrijvers, T., & Cook, W. R. (2010). EffectiveAdvice: disciplined advice with explicit effects. *Pages 109–120 of: AOSD'10: 9th International Conference on Aspect-Oriented Software Development*.

Peyton Jones, S., Vytiniotis, D., Weirich, S., & Shields, M. (2007). Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, **17**(01), 1–82.

Prehofer, C. (1997). Feature-oriented programming: A fresh look at objects. *Pages 419–443 of: ECOOP'97: 11th European Conference on Object-Oriented Programming*.

Prehofer, C. (1999). *Flexible construction of software components: A feature oriented approach*. Habilitation Thesis, Fakultät für Informatik der Technischen Universität München.

Prehofer, C. (2006). Semantic reasoning about feature composition via multiple aspect-weavings. *Pages 237–242 of: GPCE'06: 5th International conference on Generative programming and component engineering*.

Reynolds, J. C. (1974). Towards a theory of type structure. *Pages 408–423 of: Programming symposium*. Lecture Notes in Computer Science, vol. 19. Springer-Verlag.

Reynolds, John C. (1983). Types, abstraction and parametric polymorphism. *Pages 513–523 of: IFIP Congress*.

Rinard, M., Salcianu, A., & Bugrara, S. (2004). A classification system and analysis for aspect-oriented programs. *ACM SIGSOFT Software Engineering Notes*, **29**(6), 147–158.

Ruby, C., & Leavens, G. T. (2000). Safely creating correct subclasses without seeing superclass code. *Pages 208–228 of: OOPSLA'00: 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*.

Salcianu, A., & Rinard, M. C. (2005). Purity and side effect analysis for java programs. *Pages 199–215 of: VMCAI'05: 6th International Conference on Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, vol. 3385.

Schrijvers, T., & Oliveira, B. C. d. S. (2011). Monads, zippers and views: Virtualizing the monad stack. *Pages 32–44 of: ICFP'11: 16th International Conference on Functional Programming*.

Stata, R., & Guttag, J. V. (1995). Modular reasoning in the presence of subclassing. *Pages 200–214 of: OOPSLA'95: 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*.

Tanter, É. (2008). Expressive scoping of dynamically-deployed aspects. *Pages 168–179 of: AOSD'08: 7th International Conference on Aspect-Oriented Software Development*.

Voigtländer, J. (2009). Free theorems involving type constructor classes. *Pages 173–184 of: ICFP'09: 14th International Conference on Functional Programming*.

Wadler, P. (1989). Theorems for free! *Pages 347–359 of: FPLCA'89: 4th International Conference on Functional Programming and Computer Architecture*.

Wadler, P. (1992a). The essence of functional programming. *Pages 1–14 of: POPL'92: 19th Symposium on Principles of Programming Languages*.

Wadler, P. (1992b). Monads for functional programming. *Program design calculi: Marktoberdorf international summer school on program design calculi*.

Wang, M., & Oliveira, B. C. d. S. (2009). What does aspect-oriented programming mean for functional programmers? *Pages 37–48 of: Wgp'09: 8th workshop on generic programming*.

A Background on Parametricity-based Proofs

In the following sections we present a number of parametricity-based proofs related to monads and monad transformers. For this purpose, we closely follow the style and formalism set out by Voigtländer (2009), who specifically covers the technique of deriving free theorems for type constructors restricted by type class constraints.

We recommend the reader to look at Voigtländer's work for the details, but summarize the essence of the formalism here. In a nutshell, parametricity derives a *free theorem* for every expression x with a polymorphic type such as $\forall a.\tau$. This free theorem consists of a relation R between instances of the expression x_{τ_1} and x_{τ_2}. This relation R is parameterized by a relation $\mathscr{R}$ between the types τ_1 and τ_2. Typically, when a function is chosen for $\mathscr{R}$, R becomes an equational relation. Voigtländer's work captures two extensions: 1) dealing with type constructors and 2) dealing with type class constraints.

Type Constructors In Wadler's methodology for deriving free theorems (1989), free type variables are interpreted as relations between arbitrarily chosen closed types (and then quantified over via relation variables, formally denoted $\mathscr{R}$). Similarly, Voigtländer interpretes free *type constructor variables* as functions on such relations tied to arbitrarily chosen type constructors. These functions are also called *relational actions*.

Let κ_1 and κ_2 be type constructors (of kind $* \to *$). Then formally, a relational action for them, denoted $\mathscr{F} : \kappa_1 \Leftrightarrow \kappa_2$, is a function $\mathscr{F}$ on relations between closed types such that every $\mathscr{R} : \tau_1 \Leftrightarrow \tau_2$ (for arbitrary τ_1 and τ_2) is mapped to an $\mathscr{F}\,\mathscr{R} : \kappa_1\ \tau_1 \Leftrightarrow \kappa_2\ \tau_2$.

Type Class Constraints Wadler (1989) shows how to treat type class constraints for ordinary types. Basically, the relation $\mathscr{R}$ chosen as interpretation for the constrained type variable is restricted to those that relate types that are instances of the type class. Furthermore, every type class method (seen as a new constant in the language) must be related to itself by the relational interpretation.

The same approach applies to type constructor classes, where we now speak of *C actions* for relational actions restricted to type class C. For instance, for type variables constrained by the *Monad* type class, we speak of *Monad actions*. Formally, relational action $\mathscr{F} : \kappa_1 \Leftrightarrow \kappa_2$ is a Monad action iff:

- The type constructors κ_1 and κ_2 are instances of *Monad*,
- $(return_{\kappa_1}, return_{\kappa_2}) \in \forall \mathscr{R}.\mathscr{R} \to \mathscr{F}\,\mathscr{R}$, and
- $((\gg\!=_{\kappa_1}), (\gg\!=_{\kappa_2})) \in \forall \mathscr{R}.\forall \mathscr{S}.\mathscr{F}\,\mathscr{R} \to ((\mathscr{R} \to \mathscr{F}\,\mathscr{S}) \to \mathscr{F}\,\mathscr{S})$.

Monad Transformer actions and MonadState actions are defined in a similar way.

B Proof of Free Theorem for Stateful Components

Instead of the specific theorem for *new p x*, we prove a more general theorem:

Theorem 9 (STATEFUL CODE)
For any $mx :: \forall m. \mathbb{S}_{\mathrm{M}}\ s\ m \Rightarrow m\ b$ we have:

$$mx \quad \equiv \quad \begin{array}{l} \textbf{do}\ s_0 \leftarrow get \\ \quad \textbf{let}\ (r, s_1) = run\mathbb{S}\ mx\ s_0 \\ \quad put\ s_1 \\ \quad return\ r \end{array}$$

Then $mx = new\ p\ x$ is a special case.

Proof
Let $\mathscr{F} : \kappa \Leftrightarrow \mathbb{S}\ s$ be defined as

$$\mathscr{F}\ \mathscr{R} = \kappa\ \mathscr{R}; h^{-1}$$

where

$$\begin{array}{l} h\ mx = get \ggg\!\!= \lambda s_0 \rightarrow \textbf{let}\ (s_1, r) = runS\ mx\ s_0 \\ \qquad\qquad\qquad\qquad\quad \textbf{in}\ put\ s_1 \gg return\ r \end{array}$$

and ; is relation composition:

$$R_1; R_2 = \{(x,z) \mid (x,y) \in R_1, (y,z) \in R_2\}$$

We show that $\mathscr{F}$ is both a Monad and a MonadState action. Then if we choose $\mathscr{R}$ to be the identity function id, the theorem follows.

Firstly, $\mathscr{F}$ is a Monad action. Indeed,

- $(return_\kappa, return_{State\ s}) \in \mathscr{R} \rightarrow \mathscr{F}\ \mathscr{R}$ since for every $(a,b) \in \mathscr{R}$:
  - — $(return_\kappa\ a, return_\kappa\ b) \in \kappa\ \mathscr{R}$, and
  - — $(return_\kappa\ b, return_{State\ s}\ b) \in h^{-1}$, as

$$\begin{array}{ll} & get_\kappa \ggg\!\!= \lambda s_0 \rightarrow \textbf{let}\ (r_1, s_1) = run\mathbb{S}\ (return_{State\ s}\ b)\ s_0 \\ & \qquad\qquad\qquad\quad \textbf{in}\ put_\kappa\ s_1 \gg return_\kappa\ r_1 \\ \equiv & \{\text{-reduce } run\mathbb{S}\ (return_{State\ s}\ b)\ s_0\ \text{-}\} \\ & get_\kappa \ggg\!\!= \lambda s_0 \rightarrow \textbf{let}\ (r_1, s_1) = (b, s_0)\ \textbf{in}\ put_\kappa\ s_1 \gg return_\kappa\ r_1 \\ \equiv & \{\text{-eliminate let binding -}\} \\ & get_\kappa \ggg\!\!= \lambda s_0 \rightarrow put_\kappa\ s_0 \gg return_\kappa\ b \\ \equiv & \{\text{-G}\textsc{et}\text{-P}\textsc{ut}\text{ law -}\} \\ & return_\kappa\ () \gg return_\kappa\ b \\ \equiv & \{\text{-R}\textsc{eturn}\text{-B}\textsc{ind}\text{ law -}\} \\ & return_\kappa\ b \end{array}$$

- $(\ggg\!\!=_\kappa, \ggg\!\!=_{State\ s}) \in \mathscr{F}\ \mathscr{R} \rightarrow (\mathscr{R} \rightarrow \mathscr{F}\ \mathscr{S}) \rightarrow \mathscr{F}\ \mathscr{S}$, since for every $(mx_2, mx_1) \in \mathscr{F}\ \mathscr{R}$ and $(f_2, f_1) \in \mathscr{R} \rightarrow \mathscr{F}\ \mathscr{S}$ we have that $(mx_2 \ggg\!\!=_\kappa f_2, mx_1 \ggg\!\!=_{State\ s} f_1) \in \mathscr{F}\ \mathscr{S}$, as:

$$\begin{array}{l} get_\kappa \ggg\!\!= \lambda s_0 \rightarrow \textbf{let}\ (r_1, s_1) = run\mathbb{S}\ (mx_1 \ggg\!\!=_{State\ s} f_1)\ s_0 \\ \qquad\qquad\qquad\quad \textbf{in}\ put_\kappa\ s_1 \gg return_\kappa\ r_1 \end{array}$$

$$
\begin{array}{ll}
\equiv & \{\text{-unfold } \gg\!=_{\mathit{State}\ s} \text{-}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \textbf{let } (r_1,s_1) = \\
& \qquad\qquad \mathit{run}\mathbb{S}\ (\mathbb{S}\ (\lambda s_2 \to \textbf{let } (r_3,s_3) = \mathit{run}\mathbb{S}\ \mathit{mx}_1\ s_2 \\
& \qquad\qquad\qquad\qquad \textbf{in } \mathit{run}\mathbb{S}\ (f_1\ r_3)\ s_3))\ s_0 \\
& \qquad\quad \textbf{in } \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ r_1 \\
\equiv & \{\text{-reduce } \mathit{run}\mathbb{S} \text{ -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \textbf{let } (r_1,s_1) = \textbf{let } (r_3,s_3) = \mathit{run}\mathbb{S}\ \mathit{mx}_1\ s_0 \\
& \qquad\qquad\qquad\qquad \textbf{in } \mathit{run}\mathbb{S}\ (f_1\ r_3)\ s_3 \\
& \qquad\quad \textbf{in } \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ r_1 \\
\equiv & \{\text{-}\textbf{let} \text{ floating -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \textbf{let } (r_3,s_3) = \mathit{run}\mathbb{S}\ \mathit{mx}_1\ s_0 \\
& \qquad\quad \textbf{in let } (r_1,s_1) = \mathit{run}\mathbb{S}\ (f_1\ r_3)\ s_3 \\
& \qquad\qquad \textbf{in } \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ r_1 \\
\equiv & \{\text{-}\textsc{Put-Put} \text{ law -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \textbf{let } (r_3,s_3) = \mathit{run}\mathbb{S}\ \mathit{mx}_1\ s_0 \quad \textbf{in } \mathit{put}_\kappa\ s_3 \gg \\
& \qquad\quad \textbf{let } (r_1,s_1) = \mathit{run}\mathbb{S}\ (f_1\ r_3)\ s_3 \textbf{ in } \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ r_1 \\
\equiv & \{\text{-}\textsc{Return-Bind} \text{ law -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \textbf{let } (r_3,s_3) = \mathit{run}\mathbb{S}\ \mathit{mx}_1\ s_0 \quad \textbf{in } \mathit{put}_\kappa\ s_3 \gg \mathit{return}_\kappa\ s_3 \\
& \qquad \gg\!= \lambda s_4 \to \textbf{let } (r_1,s_1) = \mathit{run}\mathbb{S}\ (f_1\ r_3)\ s_4 \textbf{ in } \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ r_1 \\
\equiv & \{\text{-}\textsc{Get-Put} \text{ law -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \textbf{let } (r_3,s_3) = \mathit{run}\mathbb{S}\ \mathit{mx}_1\ s_0 \quad \textbf{in } \mathit{put}_\kappa\ s_3 \gg \mathit{get}_\kappa \gg\!= \lambda s_4 \to \\
& \qquad\quad \textbf{let } (r_1,s_1) = \mathit{run}\mathbb{S}\ (f_1\ r_3)\ s_4 \textbf{ in } \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ r_1 \\
\equiv & \{\text{-}\textsc{Return-Bind} \text{ law -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \textbf{let } (r_3,s_3) = \mathit{run}\mathbb{S}\ \mathit{mx}_1\ s_0 \quad \textbf{in } \mathit{put}_\kappa\ s_3 \gg \mathit{return}_\kappa\ r_3 \\
& \gg\!=_\kappa \lambda r_4 \to \\
& \mathit{get}_\kappa \gg\!= \lambda s_4 \to \textbf{let } (r_1,s_1) = \mathit{run}\mathbb{S}\ (f_1\ r_4)\ s_4 \textbf{ in } \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ r_1 \\
\equiv & \{\text{-fold } h \text{ -}\} \\
& h\ \mathit{mx}_1 \gg\!=_\kappa h \circ f_1
\end{array}
$$

and $(h\ \mathit{mx}_2 \gg\!=_\kappa h \circ f_2, h\ \mathit{mx}_1 \gg\!=_\kappa h \circ f_1) \in \kappa\,\mathscr{S}$.

Moreover, it is a MonadState action. Indeed,

- $(\mathit{get}_\kappa, \mathit{get}_{\mathit{State}\ s}) \in \mathscr{F}\ \mathit{id}_s$ since:

$$
\begin{array}{ll}
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \textbf{let } (r_1,s_1) = \mathit{run}\mathbb{S}\ \mathit{get}_{\mathit{State}\ s}\ s_0 \textbf{ in } \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ r_1 \\
\equiv & \{\text{-reduce } \mathit{run}\mathbb{S}\ \mathit{get}_{\mathit{State}\ s}\ s_0 \text{ -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \textbf{let } (r_1,s_1) = (s_0,s_0) \textbf{ in } \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ r_1 \\
\equiv & \{\text{-eliminate let binding -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \mathit{put}_\kappa\ s_0 \gg \mathit{return}_\kappa\ s_0 \\
\equiv & \{\text{-}\textsc{Get-Get} \text{ law -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \mathit{get}_\kappa \gg\!= \lambda s_1 \to \mathit{put}_\kappa\ s_1 \gg \mathit{return}_\kappa\ s_0 \\
\equiv & \{\text{-}\textsc{Get-Put} \text{ law -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \mathit{return}\ () \gg \mathit{return}_\kappa\ s_0 \\
\equiv & \{\text{-}\textsc{Return-Bind} \text{ law -}\} \\
& \mathit{get}_\kappa \gg\!= \lambda s_0 \to \mathit{return}_\kappa\ s_0
\end{array}
$$

$\equiv$ {-BIND-RETURN law -}
get_κ

- $(put_\kappa, put_{State\ s}) \in id_s \to \mathscr{F}\ id_{()}$ since for every s:

$get_\kappa \gg\!= \lambda s_0 \to \textbf{let}\ (r_1, s_1) = run\mathbb{S}\ (put_{State\ s}\ s)\ s_0\ \textbf{in}\ put_\kappa\ s_1 \gg return_\kappa\ r_1$
$\equiv$ {-reduce $run\mathbb{S}\ (put_{State\ s}\ s)\ s_0$ -}
$get_\kappa \gg\!= \lambda s_0 \to \textbf{let}\ (r_1, s_1) = ((), s)\ \textbf{in}\ put_\kappa\ s_1 \gg return_\kappa\ r_1$
$\equiv$ {-eliminate let binding -}
$get_\kappa \gg\!= \lambda s_0 \to put_\kappa\ s \gg return_\kappa\ ()$
$\equiv$ {-GET-QUERY law -}
$put_\kappa\ s \gg return_\kappa\ ()$
$\equiv$ {-unit singleton type -}
$put_\kappa\ s$

$\square$

C Proof of Harmlessness Theorem

We have subdivided our proof into five auxiliary lemmas. The core of the proof relies on parametricity: Lemma 2 derives the free theorem for the monad type variable of the mixin component and Lemma 3 does the same for the monad transformer type variable in the base component. However, first Lemma 1 introduces an alternative, but equivalent, shape of augmentation advice that is more suitable for deriving the free theorem, but less suitable for human consumption. Lemmas 6 and 7 simplify two complex intermediate expressions using equational reasoning, type class axioms and Lemma 2. Finally, the main proof itself in Section C.2 ties together the other four lemmas.

C.1 Auxiliary Lemmas

First, we show how to convert between the self-explanatory form of augmentation mixin 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 mix = λsuper →
    λa → mix a >>= λaft →
      super a >>= λr →
      aft r      >>= \_ →
      return r
```

Lemma 1
Consider augmentation mixin $(bef,aft) :: (a \to t\ m\ c, a \to b \to c \to t\ m\ ())$, then we have that:

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

where m is a *Monad* and t is a *MonadTrans*.

Proof

$$around\ (convert\ (bef,aft))$$
$$\equiv \{\text{-unfold } around \text{ -}\}$$
$$(\lambda super \to \lambda a \to convert\ (bef,aft)\ a \gg\!= \lambda aft'$$

$$
\begin{aligned}
&\quad\quad \to \mathit{super}\ a \gg\!= \lambda b\\
&\quad\quad \to \mathit{aft}'\ b \gg\!= \backslash\_\\
&\quad\quad \to \mathit{return}\ b)\\
\equiv\ & \{\text{-unfold } \mathit{convert} \text{ -}\}\\
&(\lambda \mathit{super} \to \lambda a \to \mathit{bef}\ a \gg\!= \lambda c\\
&\quad\quad \to \mathit{return}\ (\lambda b \to \mathit{aft}\ a\ b\ c) \gg\!= \lambda \mathit{aft}'\\
&\quad\quad \to \mathit{super}\ a \qquad\qquad \gg\!= \lambda b\\
&\quad\quad \to \mathit{aft}'\ b \gg\!= \backslash\_\\
&\quad\quad \to \mathit{return}\ b)\\
\equiv\ & \{\text{-}\textsc{Return-Bind}\text{ law -}\}\\
&(\lambda \mathit{super} \to \lambda a \to \mathit{bef}\ a \gg\!= \lambda c\\
&\quad\quad \to \mathit{super}\ a \qquad\qquad \gg\!= \lambda b\\
&\quad\quad \to \mathit{aft}\ a\ b\ c \gg\!= \backslash\_\\
&\quad\quad \to \mathit{return}\ b)\\
\equiv\ & \{\text{-fold } \mathit{augment} \text{ -}\}\\
&\mathit{augment}\ (\mathit{bef}, \mathit{aft})
\end{aligned}
$$

□

Here is the second auxiliary lemma.

Lemma 2
Consider a function $f :: \forall m.\mathit{Monad}\ m \Rightarrow a \to m\ (b \to m\ c)$, then we have that:

$$f_m \equiv (\mathit{out}\ (\mathit{return} \circ \mathit{out}\ (\mathit{return} \circ \mathit{run}\mathbb{I}) \circ \mathit{run}\mathbb{I}))\ f_{\mathbb{I}}$$

Proof
Let $\mathscr{F} : m \Leftrightarrow \mathbb{I}$ be the *Monad* action

$$\mathscr{F}\ \mathscr{R} = \mathit{return}^{-1}; \mathscr{R}; \mathbb{I}$$

. 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). □

Lemma 3
Consider a function $f :: \forall t.\mathit{MonadTrans}\ t \Rightarrow (a \to t\ m\ (b \to t\ m\ ())) \to a \to t\ m\ b$ with m an arbitrary monad, then we have that:

$$\begin{gathered}\mathit{out}\ \pi \circ f\ \equiv\\ \mathit{out}\ \mathit{run}\mathbb{I}_{\mathrm{T}} \circ f \circ \mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \pi)) \circ \pi)\end{gathered}$$

for any $\pi :: \forall m, a.\mathit{Monad}\ m \Rightarrow t\ m\ a \to m\ a$ with t an arbitrary monad transformer that satisfies the following property:

$$\pi \circ \mathit{lift} \quad \equiv \quad \mathit{id}$$

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

Proof

Let $\mathscr{T} : \tau \Leftrightarrow \mathbb{I}_\mathrm{T}$ be the *MonadTrans* action

$$\mathscr{T}\ \mathscr{F}\ \mathscr{R} = \pi; \mathscr{F}\ \mathscr{R}; \mathbb{I}_\mathrm{T}$$

. This is a *MonadTrans* action indeed:

- $(\mathit{lift}_t, \mathit{lift}_{\mathbb{I}_\mathrm{T}}) \in \forall \mathscr{F}, \mathscr{R}. \mathscr{F}\ \mathscr{R} \to \mathscr{T}\ \mathscr{F}\ \mathscr{R}$, since for every $(a,b) \in \mathscr{F}\ \mathscr{R}$ we have $(\mathit{lift}_t\ a, \mathit{lift}_{\mathbb{I}_\mathrm{T}}\ b) = (\mathit{lift}_t\ a, \mathbb{I}_\mathrm{T}\ b) \in \pi; \mathscr{F}\ \mathscr{R}; \mathbb{I}_\mathrm{T}$ because of Property (1) of π.

Then we have for all $(h,h') \in (id_b \to \mathscr{T}\ m\ id_{()})$, that $h' \equiv \mathbb{I}_\mathrm{T} \circ \pi \circ h)$, because $m\ id \equiv id$. Assume that $(g,g') \in id_a \to \mathscr{T}\ m\ (id_b \to \mathscr{T}\ m\ id_{()})$, where $g' \equiv \mathit{out}\ (\mathbb{I}_\mathrm{T} \circ \mathit{fmap}\ (\mathit{out}\ (\mathbb{I}_\mathrm{T} \circ \pi)) \circ \pi)\ g$. Then, for $(f\ g, f\ g') \in id_a \to \mathscr{T}\ m\ id_b$ the lemma follows.

Now, we only have to show that the assumption wrt. (g,g') is valid. The assumption is valid if for all $(a,a) \in id_a$, we have that $(ga, g'a) \in \mathscr{T}\ m\ (id_b \to \mathscr{T}\ m\ id_{()})$. This holds if, applying $\mathscr{T}$, we have that $(proj(ga), unIdT(g'a)) \in m\ (id_b \to \mathscr{T}\ m\ id_{()})$. By equational reasoning, we get

$$\begin{array}{l}
\quad (\pi\ (g\ a), run\mathbb{I}_\mathrm{T}\ (g'\ a)) \\
\equiv\ \ \{\text{-unfold } g' \text{ -}\} \\
\quad (\pi\ (g\ a), run\mathbb{I}_\mathrm{T} \circ \mathbb{I}_\mathrm{T} \circ \mathit{fmap}\ (\mathit{out}\ (\mathbb{I}_\mathrm{T} \circ \pi))\ \$\ \pi\ (g\ a)) \\
\equiv\ \ \{\text{-}run\mathbb{I}_\mathrm{T} \circ \mathbb{I}_\mathrm{T} \equiv id\text{ -}\} \\
\quad (\pi\ (g\ a), \mathit{fmap}\ (\mathit{out}\ (\mathbb{I}_\mathrm{T} \circ \pi))\ \$\ \pi\ (g\ a)) \\
\equiv\ \ \{\text{-unfold } \mathit{fmap} \text{ and } \mathit{out} \text{ -}\} \\
\quad (\pi\ (g\ a), \pi\ (g\ a) \gg\!= \lambda f \to \mathit{return}\ (\mathbb{I}_\mathrm{T} \circ \pi \circ f)) \\
\equiv\ \ \{\text{-}\textsc{Bind-Return}\text{ law -}\} \\
\quad (\pi\ (g\ a) \gg\!= \mathit{return}, \\
\quad\quad \pi\ (g\ a) \gg\!= \lambda f \to \mathit{return}\ (\mathbb{I}_\mathrm{T} \circ \pi \circ f))
\end{array}$$

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

Here is the fourth auxiliary lemma.

Lemma 6
Consider a function $mix :: \forall m.\mathit{Monad}\ m \Rightarrow a \to t\ m\ (b \to t\ m\ ())$, then we have that:

$$\begin{array}{c}
(\mathit{out}\ (\mathbb{I}_\mathrm{T} \circ \mathit{fmap}\ (\mathit{out}\ (\mathbb{I}_\mathrm{T} \circ \pi)) \circ \pi))\ mix \\
\equiv \\
\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))
\end{array}$$

where t is a *MonadTrans*.

Proof

$$\begin{array}{l}
\quad (\mathit{out}\ (\mathbb{I}_\mathrm{T} \circ \mathit{fmap}\ (\mathit{out}\ (\mathbb{I}_\mathrm{T} \circ \pi)) \circ \pi))\ mix \\
\equiv\ \ \{\text{-}\mathit{out}\ (f \circ g) \equiv \mathit{out} f \circ \mathit{out}\ g\text{ -}\}
\end{array}$$

$$
\begin{array}{ll}
& (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}} \circ \mathit{out}\ \pi) \circ \pi))\ \mathit{mix} \\
\equiv & \{\text{-}\mathit{fmap}\ (g \circ h) \equiv \mathit{fmap}\ g \circ \mathit{fmap}\ h\ \text{-}\} \\
& (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}}) \circ \mathit{fmap}\ (\mathit{out}\ \pi) \circ \pi))\ \mathit{mix} \\
\equiv & \{\text{-}\mathit{out}\ (f \circ g) \equiv \mathit{out}\ f \circ \mathit{out}\ g\ \text{-}\} \\
& (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}})) \circ \mathit{out}\ (\mathit{fmap}\ (\mathit{out}\ \pi) \circ \pi))\ \mathit{mix} \\
\equiv & \{\text{-unfold def. of } (\circ)\ \text{-}\} \\
& (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}})))\ (\mathit{out}\ (\mathit{fmap}\ (\mathit{out}\ \pi) \circ \pi)\ \mathit{mix}) \\
\equiv & \{\text{-Lemma 2 -}\} \\
& (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}})))\ ((\mathit{out}\ (\mathit{return} \circ \mathit{out}\ (\mathit{return} \circ \mathit{run}\mathbb{I}) \circ \mathit{run}\mathbb{I})) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (\mathit{out}\ (\mathit{fmap}\ (\mathit{out}\ \pi) \circ \pi)\ \mathit{mix})) \\
\equiv & \{\text{-}\mathit{out}\ (f \circ g) \equiv \mathit{out}\ f \circ \mathit{out}\ g\ (\times 3)\ \text{-}\} \\
& (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}})))\ ((\mathit{out}\ (\mathit{return} \circ \mathit{out}\ \mathit{return}) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (\mathit{out}\ (\mathit{out}\ \mathit{run}\mathbb{I} \circ \mathit{run}\mathbb{I})\ (\mathit{out}\ (\mathit{fmap}\ (\mathit{out}\ \pi) \circ \pi)\ \mathit{mix}))) \\
\equiv & \{\text{-Totality assumption -}\} \\
& (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}})))\ ((\mathit{out}\ (\mathit{return} \circ \mathit{out}\ \mathit{return}))\ (\mathit{const}\ (\mathit{const}\ ()))) \\
\equiv & \{\text{-unfold def. of } (\circ) \text{ and } \mathit{out}\ \text{-}\} \\
& (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}})))\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))) \\
\equiv & \{\text{-unfold def. of } \mathit{out}\ \text{-}\} \\
& \mathit{const}\ ((\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}}) \circ \mathit{return})\ (\mathit{const}\ (\mathit{return}\ ()))) \\
\equiv & \{\text{-}\mathit{fmap}\ h \circ \mathit{return} \equiv \mathit{return} \circ h\ \text{-}\} \\
& \mathit{const}\ ((\mathbb{I}_{\mathrm{T}} \circ \mathit{return} \circ \mathit{out}\ \mathbb{I}_{\mathrm{T}})\ (\mathit{const}\ (\mathit{return}\ ()))) \\
\equiv & \{\text{-}\mathbb{I}_{\mathrm{T}} \circ \mathit{return} \equiv \mathit{return}\ \text{-}\} \\
& \mathit{const}\ ((\mathit{return} \circ \mathit{out}\ \mathbb{I}_{\mathrm{T}})\ (\mathit{const}\ (\mathit{return}\ ()))) \\
\equiv & \{\text{-}\mathit{out}\ f\ (\mathit{const}\ x) \equiv \mathit{const}\ (f\ x)\ \text{-}\} \\
& \mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathbb{I}_{\mathrm{T}}\ (\mathit{return}\ ())))) \\
\equiv & \{\text{-}\mathbb{I}_{\mathrm{T}} \circ \mathit{return} \equiv \mathit{return}\ \text{-}\} \\
& \mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))
\end{array}
$$

$\square$

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

$$
\begin{array}{l}
(\otimes) :: \forall t\ m\ a\ b.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \\
\quad \Rightarrow \mathit{Augment}\ a\ t\ m\ b \\
\quad \rightarrow \mathit{Open}\ (a \rightarrow t\ m\ b) \\
\quad \rightarrow \mathit{Open}\ (a \rightarrow t\ m\ b) \\
\mathit{mixin} \otimes \mathit{base} = \mathit{around}\ \mathit{mixin} \ominus \mathit{base}
\end{array}
$$

Here is the fifth auxiliary lemma.

Lemma 7
Consider a function $bse :: \forall t. MonadTrans\ t \Rightarrow Open\ (a \rightarrow t\ m\ b)$, then we have that:

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

where m is a *Monad*.

Proof

$$
\begin{aligned}
& new\ ((const\ (return\ (const\ (return\ ())))) \otimes bse) \\
\equiv\ & \{\text{-unfold def. of } \otimes \text{ -}\} \\
& new\ (\lambda p\ x \rightarrow const\ (return\ (const\ (return\ ())))\ x \gg\!= \lambda aft \rightarrow bse\ p\ x \\
& \qquad \gg\!= \lambda r \rightarrow aft\ r \gg\!= \backslash\_ \rightarrow return\ r) \\
\equiv\ & \{\text{-unfold } const \text{ -}\} \\
& new\ (\lambda p\ x \rightarrow return\ (const\ (return\ ())) \gg\!= \lambda aft \rightarrow bse\ p\ x \\
& \qquad \gg\!= \lambda r \rightarrow aft\ r \gg\!= \backslash\_ \rightarrow return\ r) \\
\equiv\ & \{\text{-RETURN-BIND law -}\} \\
& new\ (\lambda p\ x \rightarrow bse\ p\ x \gg\!= \lambda r \rightarrow const\ (return\ ())\ r \gg\!= \backslash\_ \rightarrow return\ r) \\
\equiv\ & \{\text{-unfold } const \text{ -}\} \\
& new\ (\lambda p\ x \rightarrow bse\ p\ x \gg\!= \lambda r \rightarrow return\ () \gg\!= \backslash\_ \rightarrow return\ r) \\
\equiv\ & \{\text{-RETURN-BIND law -}\} \\
& new\ (\lambda p\ x \rightarrow bse\ p\ x \gg\!= \lambda r \rightarrow return\ r) \\
\equiv\ & \{\text{-}\eta\text{-reduction -}\} \\
& new\ (\lambda p\ x \rightarrow bse\ p\ x \gg\!= return) \\
\equiv\ & \{\text{-BIND-RETURN law -}\} \\
& new\ (\lambda p\ x \rightarrow bse\ p\ x) \\
\equiv\ & \{\text{-}\eta\text{-reduction -}\} \\
& new\ bse
\end{aligned}
$$

□

C.2 Main Proof

The main theorem follows from the above lemmas.

Proof

$$
\begin{aligned}
& \pi \circ new\ ((bef, aft) \circledast bse) \\
\equiv\ & \{\ \text{Lemma 1}\ \} \\
& \pi \circ new\ (convert\ (bef, aft) \otimes bse) \\
\equiv\ & \{\ \textbf{let}\ mix = convert\ (bef, aft)\ \} \\
& \pi \circ new\ (mix \otimes bse) \\
\equiv\ & \{\ \text{abstract over } mix\ \} \\
& \pi \circ ((\lambda x \rightarrow new\ (x \otimes bse))\ mix) \\
\equiv\ & \{\ \text{fold } out\ \} \\
& (out\ \pi \circ (\lambda x \rightarrow new\ (x \otimes bse)))\ mix
\end{aligned}
$$

$\equiv$ { Lemma 3 }
$(out\ run\mathbb{I}_T \circ (\lambda x \to new\ (x \otimes bse)) \circ out\ (\mathbb{I}_T \circ fmap\ (out\ (\mathbb{I}_T \circ \pi)) \circ \pi))\ mix$
$\equiv$ { unfold def. of $(\circ)$ }
$(out\ run\mathbb{I}_T \circ (\lambda x \to new\ (x \otimes bse)))\ ((out\ (\mathbb{I}_T \circ fmap\ (out\ (\mathbb{I}_T \circ \pi)) \circ \pi))\ mix)$
$\equiv$ { Lemma 6 }
$(out\ run\mathbb{I}_T \circ (\lambda x \to new\ (x \otimes bse)))\ (const\ (return\ (const\ (return\ ()))))$
$\equiv$ { unfold def. of $(\circ)$ }
$out\ run\mathbb{I}_T\ ((\lambda x \to new\ (x \otimes bse))\ (const\ (return\ (const\ (return\ ())))))$
$\equiv$ { β-reduction }
$out\ run\mathbb{I}_T\ (new\ ((const\ (return\ (const\ (return\ ())))) \otimes bse))$
$\equiv$ { Lemma 7 }
$out\ run\mathbb{I}_T\ (new\ bse)$

□

D Proofs of Projection Functions

In this section we prove the projection function precondition of the Harmless Mixin theorem for two specific monads. The proofs are fairly straightforward equational reasoning and application of the monad axioms.

D.1 The π_W Function

Proof

$\pi_W \circ lift$
$\equiv$ {-unfold $\circ$ -}
$(\lambda m \to \pi_W\ (lift\ m))$
$\equiv$ {-unfold *lift* -}
$(\lambda m \to \pi_W\ (\mathbb{W}_T\ (m \gg\!= \lambda x \to return\ (x, mempty))))$
$\equiv$ {-unfold π_W -}
$(\lambda m \to run\mathbb{W}_T\ (\mathbb{W}_T\ (m \gg\!= \lambda x \to return\ (x, mempty))) \gg\!= return \circ fst)$
$\equiv$ {-$run\mathbb{W}_T\ (\mathbb{W}_T\ m) \equiv m$ -}
$(\lambda m \to m \gg\!= \lambda x \to return\ (x, mempty) \gg\!= return \circ fst)$
$\equiv$ {-Return-Bind law -}
$(\lambda m \to m \gg\!= \lambda x \to (return \circ fst)\ (x, mempty))$
$\equiv$ {-unfold $\circ$ -}
$(\lambda m \to m \gg\!= \lambda x \to return\ (fst\ (x, mempty)))$
$\equiv$ {-unfold *fst* -}
$(\lambda m \to m \gg\!= \lambda x \to return\ x)$
$\equiv$ {-η-reduction -}
$(\lambda m \to m \gg\!= return)$
$\equiv$ {-Bind-Return law -}
$(\lambda m \to m)$
$\equiv$ {-fold *id* -}
id

□

D.2 The π_S Function

Proof

$$
\begin{array}{ll}
& \pi_S\ s_0 \circ \mathit{lift} \\
\equiv & \{\text{-unfold} \circ \text{-}\} \\
& (\lambda m \to \pi_S\ s_0\ (\mathit{lift}\ m)) \\
\equiv & \{\text{-unfold } \mathit{lift} \text{ -}\} \\
& (\lambda m \to \pi_S\ s_0\ (\mathbb{S}_{\mathrm{T}}\ (\lambda s \to m \gg\!= \lambda x \to \mathit{return}\ (x,s)))) \\
\equiv & \{\text{-unfold } \pi_S \text{ -}\} \\
& (\lambda m \to \mathit{run}\mathbb{S}_{\mathrm{T}}\ (\mathbb{S}_{\mathrm{T}}\ (\lambda s \to m \gg\!= \lambda x \to \mathit{return}\ (x,s)))\ s_0 \gg\!= \mathit{return} \circ \mathit{fst}) \\
\equiv & \{\text{-}\mathit{run}\mathbb{S}_{\mathrm{T}}\ (\mathbb{S}_{\mathrm{T}}\ f) \equiv f \text{ -}\} \\
& (\lambda m \to (\lambda s \to m \gg\!= \lambda x \to \mathit{return}\ (x,s))\ s_0 \gg\!= \mathit{return} \circ \mathit{fst}) \\
\equiv & \{\text{-}\beta\text{-reduction -}\} \\
& (\lambda m \to m \gg\!= \lambda x \to \mathit{return}\ (x,s_0) \gg\!= \mathit{return} \circ \mathit{fst}) \\
\equiv & \{\text{-R}\text{ETURN-}\text{B}\text{IND law -}\} \\
& (\lambda m \to m \gg\!= \lambda x \to (\mathit{return} \circ \mathit{fst})\ (x,s_0)) \\
\equiv & \{\text{-unfold} \circ \text{-}\} \\
& (\lambda m \to m \gg\!= \lambda x \to \mathit{return}\ (\mathit{fst}\ (x,s_0))) \\
\equiv & \{\text{-unfold } \mathit{fst} \text{ -}\} \\
& (\lambda m \to m \gg\!= \lambda x \to \mathit{return}\ x) \\
\equiv & \{\text{-}\eta\text{-reduction -}\} \\
& (\lambda m \to m \gg\!= \mathit{return}) \\
\equiv & \{\text{-B}\text{IND-}\text{R}\text{ETURN law -}\} \\
& (\lambda m \to m) \\
\equiv & \{\text{-fold } \mathit{id} \text{ -}\} \\
& \mathit{id}
\end{array}
$$

□

E Proof of Harmless Observation Mixin

The structure of the Harmless Observation Mixin proof is the same as that of the Harmless Mixin proof. We can even reuse two of the five lemmas. The other three need to be adjusted to the *MGet* additional constraint. Notably, Lemma 8 derives the corresponding parametricity result which is weaker than that of Lemma 2. Fortunately, Lemma 9 compensates for this by exploiting the *MGet* axioms.

E.1 Auxiliary Lemmas

Again we turn to the same convenient intermediate form for augmentation mixins that we used for the proof of orthogonal harmless mixins. We define ⊚ as the counterpart of ⊙ for:

$$
\begin{array}{l}
(\circledcirc) :: (\mathit{MGet}\ s\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \\
\quad \Rightarrow (a \to t\ m\ (b \to t\ m\ ()))
\end{array}
$$

$\quad \rightarrow Open\ (a \rightarrow t\ m\ b)$
$\quad \rightarrow Open\ (a \rightarrow t\ m\ b)$
$mixin \circledcirc base = around\ mixin\ `observation`\ base$

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

Lemma 8
Consider a function $f :: \forall m.MGet\ s\ m \Rightarrow a \rightarrow m\ (b \rightarrow m\ c)$, then we have that:

$$f_m \equiv (out\ (\lambda m \rightarrow aux\ m \gg\!= return \circ out\ aux))\ f_{(\mathbb{R}\ s)}$$

where

$aux :: MGet\ s\ m \Rightarrow \mathbb{R}\ s\ a \rightarrow m\ a$
$aux\ m = get \gg\!= \lambda s \rightarrow return\ (run\mathbb{R}\ m\ s)$

Proof
Let $\mathscr{F} : m \Leftrightarrow \mathbb{R}\ s$ be the *MGet s* action

$$\mathscr{F}\ \mathscr{R} = (get \gg\!=)^{-1}\ ;\ out\ return^{-1}\ ;\ id_s \rightarrow \mathscr{R}\ ;\ \mathbb{R}$$

.

This is indeed a *MGet s* action:

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

$$\begin{array}{ll}
 & get \gg\!= \lambda s \rightarrow return\ (f_1\ s) \\
\equiv & \{\text{-unfold } return \text{ -}\} \\
 & get \gg\!= \lambda s \rightarrow \mathbb{R}\ (const\ (f_1\ s)) \\
\equiv & \{\text{-unfold } get \text{ -}\} \\
 & \mathbb{R}\ id \gg\!= \lambda s \rightarrow \mathbb{R}\ (const\ (f_1\ s)) \\
\equiv & \{\text{-unfold } \gg\!= \text{ -}\} \\
 & \mathbb{R}\ \$\ \lambda s \rightarrow run\mathbb{R}\ (\mathbb{R}\ (const\ (f_1\ (run\mathbb{R}\ (\mathbb{R}\ id)\ s))))\ s \\
\equiv & \{\text{-}run\mathbb{R}\ (\mathbb{R}\ f) \equiv f \text{ -}\} \\
 & \mathbb{R}\ \$\ \lambda s \rightarrow const\ (f_1\ (id\ s))\ s \\
\equiv & \{\text{-unfold } const \text{ -}\} \\
 & \mathbb{R}\ \$\ \lambda s \rightarrow f_1\ (id\ s) \\
\equiv & \{\text{-unfold } id \text{ -}\} \\
 & \mathbb{R}\ \$\ \lambda s \rightarrow f_1\ s
\end{array}$$

$$\begin{array}{ll} \equiv & \{\text{-}\eta\text{-reduction -}\} \\ & \mathbb{R}\, f_1 \end{array}$$

Similarly, we can show that

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

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

$$\begin{array}{ll} & get \gg\!= \lambda s \to return\ (f_1\ s) \gg\!= \lambda x \to get \gg\!= \lambda s' \to return\ (k_1\ x\ s') \\ \equiv & \{\text{-}\textsc{Return-Bind}\text{ law -}\} \\ & get \gg\!= \lambda s \to get \gg\!= \lambda s' \to return\ (k_1\ (f_1\ s)\ s') \\ \equiv & \{\text{-get idempotence -}\} \\ & get \gg\!= \lambda s \to return\ (k_1\ (f_1\ s)\ s) \end{array}$$

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

$$\begin{array}{ll} & \mathbb{R}\, f_2 \gg\!= \lambda x \to \mathbb{R}\ (k_2\ x) \\ \equiv & \{\text{-unfold} \gg\!= \text{-}\} \\ & \mathbb{R}\ \$\ \lambda s \to run\mathbb{R}\ (\mathbb{R}\ (k_2\ (run\mathbb{R}\ (\mathbb{R}\, f_2)\ s)))\ s \\ \equiv & \{\text{-}run\mathbb{R}\ (\mathbb{R}\, f) \equiv f\text{ -}\} \\ & \mathbb{R}\ \$\ \lambda s \to run\mathbb{R}\ (\mathbb{R}\ (k_2\ (f_2\ s)))\ s \\ \equiv & \{\text{-}run\mathbb{R}\ (\mathbb{R}\, f) \equiv f\text{ -}\} \\ & \mathbb{R}\ \$\ \lambda s \to k_2\ (f_2\ s)\ s \end{array}$$

Summarizing, the original pair is in $\mathscr{F}\ \mathscr{S}$. Hence, we have that $(\gg\!=_m, \gg\!=_{\mathbb{R}\, s}) \in \mathscr{F}\ \mathscr{R} \to (\mathscr{R} \to \mathscr{F}\ \mathscr{S}) \to \mathscr{F}\ \mathscr{S}$.

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

The next lemma is the counterpart of Lemma 6.

Lemma 9
Consider a function $mix :: \forall m. MGet\ s\ m \Rightarrow a \to t\ m\ (b \to t\ m\ ())$, then we have that:

$$\begin{array}{c} (out\ (\mathbb{I}_{\mathrm{T}} \circ fmap\ (out\ (\mathbb{I}_{\mathrm{T}} \circ \pi)) \circ \pi))\ mix \\ \equiv \\ const\ (return\ (const\ (return\ ()))) \end{array}$$

where t is a *MonadTrans*.

Proof

$$\begin{array}{ll} & (out\ (\mathbb{I}_{\mathrm{T}} \circ fmap\ (out\ (\mathbb{I}_{\mathrm{T}} \circ \pi)) \circ \pi))\ mix \\ \equiv & \{\text{-}out\ (f \circ g) \equiv out\, f \circ out\ g\text{ -}\} \\ & (out\ (\mathbb{I}_{\mathrm{T}} \circ fmap\ (out\ \mathbb{I}_{\mathrm{T}} \circ out\ \pi) \circ \pi))\ mix \end{array}$$

$\equiv$ {-$\mathit{fmap}\ (g \circ h) \equiv \mathit{fmap}\ g \circ \mathit{fmap}\ h$ -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T) \circ \mathit{fmap}\ (\mathit{out}\ \pi) \circ \pi))\ \mathit{mix}$
$\equiv$ {-$\mathit{out}\ (f \circ g) \equiv \mathit{out}\ f \circ \mathit{out}\ g$ -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)) \circ \mathit{out}\ (\mathit{fmap}\ (\mathit{out}\ \pi) \circ \pi))\ \mathit{mix}$
$\equiv$ {-unfold def. of $(\circ)$ -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ (\mathit{out}\ (\mathit{fmap}\ (\mathit{out}\ \pi) \circ \pi)\ \mathit{mix})$
$\equiv$ {-**let** $\mathit{adv}' = (\mathit{out}\ (\mathit{fmap}\ (\mathit{out}\ \pi) \circ \pi)\ \mathit{mix})$ -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ \mathit{adv}'$
$\equiv$ {-Lemma 8 -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ ((\mathit{out}\ (\lambda m \to \mathit{aux}\ m \gg\!= \mathit{return} \circ \mathit{out}\ \mathit{aux}))\ \mathit{adv}')$
$\equiv$ {-unfold aux -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ ((\mathit{out}\ (\lambda m \to \mathit{aux}\ m$
$\quad \gg\!= \mathit{return} \circ \mathit{out}\ (\lambda n \to \mathit{get} \gg\!= \lambda s \to \mathit{return}\ (\mathit{run}\mathbb{R}\ n\ s))))\ \mathit{adv}')$
$\equiv$ {-Totality assumption -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ ((\mathit{out}\ (\lambda m \to \mathit{aux}\ m$
$\quad \gg\!= \mathit{return} \circ \mathit{out}\ (\lambda n \to \mathit{get} \gg\!= \lambda s \to \mathit{return}\ ())))\ \mathit{adv}')$
$\equiv$ {-GET-QUERY law -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ ((\mathit{out}\ (\lambda m \to \mathit{aux}\ m$
$\quad \gg\!= \mathit{return} \circ \mathit{out}\ (\lambda n \to \mathit{return}\ ())))\ \mathit{adv}')$
$\equiv$ {-fold const -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ ((\mathit{out}\ (\lambda m \to \mathit{aux}\ m$
$\quad \gg\!= \mathit{return} \circ \mathit{out}\ (\mathit{const}\ (\mathit{return}\ ()))))\ \mathit{adv}')$
$\equiv$ {-unfold aux -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ ((\mathit{out}\ (\lambda m \to \mathit{get} \gg\!= \lambda s \to \mathit{return}\ (\mathit{run}\mathbb{R}\ m\ s)$
$\quad \gg\!= \mathit{return} \circ \mathit{out}\ (\mathit{const}\ (\mathit{return}\ ()))))\ \mathit{adv}')$
$\equiv$ {-RETURN-BIND law -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))$
$\quad ((\mathit{out}\ (\lambda m \to \mathit{get} \gg\!= \lambda s \to \mathit{return}\ (\mathit{out}\ (\mathit{const}\ (\mathit{return}\ ()))\ (\mathit{run}\mathbb{R}\ m\ s))))\ \mathit{adv}')$
$\equiv$ {-$\mathit{out}\ (\mathit{const}\ x)\ y \equiv \mathit{const}\ y$ -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))$
$\quad ((\mathit{out}\ (\lambda m \to \mathit{get} \gg\!= \lambda s \to \mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))\ \mathit{adv}')$
$\equiv$ {-GET-QUERY law -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ ((\mathit{out}\ (\lambda m \to \mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))\ \mathit{adv}')$
$\equiv$ {-fold const -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ ((\mathit{out}\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))))\ \mathit{adv}')$
$\equiv$ {-$\mathit{out}\ (\mathit{const}\ x)\ y \equiv \mathit{const}\ y$ -}
$(\mathit{out}\ (\mathbb{I}_T \circ \mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))$
$\equiv$ {-$\mathit{out}\ (f \circ g) \equiv \mathit{out}\ f \circ \mathit{out}\ g$ -}
$(\mathit{out}\ \mathbb{I}_T \circ \mathit{out}\ (\mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)))\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))$
$\equiv$ {-unfold $\circ$ -}
$\mathit{out}\ \mathbb{I}_T\ (\mathit{out}\ (\mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T))\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))))$
$\equiv$ {-out f (const x) == const (f x) -}
$\mathit{out}\ \mathbb{I}_T\ (\mathit{const}\ (\mathit{fmap}\ (\mathit{out}\ \mathbb{I}_T)\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))))$
$\equiv$ {-$\mathit{fmap}\ f\ (\mathit{return}\ x) = \mathit{return}\ (f\ x)$ -}

$$
\begin{aligned}
&\quad \mathit{out}\ \mathbb{I}_{\mathrm{T}}\ (\mathit{const}\ (\mathit{return}\ (\mathit{out}\ \mathbb{I}_{\mathrm{T}}\ (\mathit{const}\ (\mathit{return}\ ()))))) \\
&\equiv \{\text{-}\mathit{out}\,f\ (\mathit{const}\ x) \equiv \mathit{const}\ (f\ x)\ \text{-}\} \\
&\quad \mathit{out}\ \mathbb{I}_{\mathrm{T}}\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathbb{I}_{\mathrm{T}}\ (\mathit{return}\ ()))))) \\
&\equiv \{\text{-}\mathbb{I}_{\mathrm{T}}\ (\mathit{return}\ x) \equiv \mathit{return}\ x\ \text{-}\} \\
&\quad \mathit{out}\ \mathbb{I}_{\mathrm{T}}\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))) \\
&\equiv \{\text{-out f (const x) == const (f x) -}\} \\
&\quad \mathit{const}\ (\mathbb{I}_{\mathrm{T}}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))) \\
&\equiv \{\text{-}\mathbb{I}_{\mathrm{T}}\ (\mathit{return}\ x) \equiv \mathit{return}\ x\ \text{-}\} \\
&\quad \mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))
\end{aligned}
$$

□

Lemma 10
Consider a function $\mathit{bse} :: \forall t.\mathit{MonadTrans}\ t \Rightarrow \mathit{Open}\ (a \to t\ m\ b)$, then we have that:

$$
\mathit{new}\ ((\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))) \circledcirc \mathit{bse}) \\ \equiv \\ \mathit{new}\ \mathit{bse}
$$

where m is a *Monad*.

Proof

$$
\begin{aligned}
&\quad \mathit{new}\ ((\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))) \circledcirc \mathit{bse}) \\
&\equiv \{\text{-unfold def. of } \circledcirc \text{ -}\} \\
&\quad \mathit{new}\ (\lambda p\ x \to \mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))\ x \gg\!= \lambda \mathit{aft} \to \mathit{bse}\ p\ x \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \lambda r \to \mathit{aft}\ r \gg\!= \backslash\_ \to \mathit{return}\ r) \\
&\equiv \{\text{-fold def. of } \otimes \text{ -}\} \\
&\quad \mathit{new}\ ((\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))) \otimes \mathit{bse}) \\
&\equiv \{\text{-Lemma 7 -}\} \\
&\quad \mathit{new}\ \mathit{bse}
\end{aligned}
$$

□

E.2 Main proof

The main proof is similar to the Harmless Mixin proof. The only difference lies in the use of Lemma 9, which relies on the GET-QUERY law.

Proof

$$
\begin{aligned}
&\quad \pi \circ \mathit{new}\ ((\mathit{bef}, \mathit{aft}) \odot \mathit{bse}) \\
&\equiv \{\text{-Lemma 1 -}\} \\
&\quad \pi \circ \mathit{new}\ (\mathit{convert}\ (\mathit{bef}, \mathit{aft}) \circledcirc \mathit{bse}) \\
&\equiv \{\text{-}\mathbf{let}\ \mathit{mix} = \mathit{convert}\ (\mathit{bef}, \mathit{aft})\ \text{-}\} \\
&\quad \pi \circ \mathit{new}\ (\mathit{mix} \circledcirc \mathit{bse}) \\
&\equiv \{\text{-abstract over } \mathit{mix} \text{ -}\} \\
&\quad \pi \circ ((\lambda x \to \mathit{new}\ (x \circledcirc \mathit{bse}))\ \mathit{mix})
\end{aligned}
$$

$\equiv$ {-fold *out* -}
$(\mathit{out}\ \pi \circ (\lambda x \rightarrow \mathit{new}\ (x \circledcirc \mathit{bse})))\ \mathit{mix}$
$\equiv$ {-Lemma 3 -}
$(\mathit{out}\ \mathit{run}\mathbb{I}_{\mathrm{T}} \circ (\lambda x \rightarrow \mathit{new}\ (x \circledcirc \mathit{bse})) \circ \mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \pi)) \circ \pi))\ \mathit{mix}$
$\equiv$ {-unfold def. of $(\circ)$ -}
$(\mathit{out}\ \mathit{run}\mathbb{I}_{\mathrm{T}} \circ (\lambda x \rightarrow \mathit{new}\ (x \circledcirc \mathit{bse})))\ ((\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \mathit{fmap}\ (\mathit{out}\ (\mathbb{I}_{\mathrm{T}} \circ \pi)) \circ \pi))\ \mathit{mix})$
$\equiv$ {-Lemma 9 -}
$(\mathit{out}\ \mathit{run}\mathbb{I}_{\mathrm{T}} \circ (\lambda x \rightarrow \mathit{new}\ (x \circledcirc \mathit{bse})))\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ()))))$
$\equiv$ {-unfold def. of $(\circ)$ -}
$\mathit{out}\ \mathit{run}\mathbb{I}_{\mathrm{T}}\ ((\lambda x \rightarrow \mathit{new}\ (x \circledcirc \mathit{bse}))\ (\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))))$
$\equiv$ {-β-reduction -}
$\mathit{out}\ \mathit{run}\mathbb{I}_{\mathrm{T}}\ (\mathit{new}\ ((\mathit{const}\ (\mathit{return}\ (\mathit{const}\ (\mathit{return}\ ())))) \circledcirc \mathit{bse}))$
$\equiv$ {-Lemma 10 -}
$\mathit{out}\ \mathit{run}\mathbb{I}_{\mathrm{T}}\ (\mathit{new}\ \mathit{bse})$

□