Under consideration for publication in J. Functional Programming

Interleaving data and effects

ROBERT ATKEY, PATRICIA JOHANN, NEIL GHANI
University of Strathclyde, Glasgow, G1 1XH, UK

BART JACOBS
Radboud University, Nijmegen, Netherlands
(*e-mail:* {robert.atkey,patricia.johann,neil.ghani}@strath.ac.uk, bart@cs.ru.nl)

Abstract

The study of programming with and reasoning about inductive datatypes such as lists and trees has benefited from the simple categorical principle of initial algebras. In initial algebra semantics, each inductive datatype is represented by an initial f-algebra for an appropriate functor f. The initial algebra principle then supports the straightforward derivation of definitional principles and proof principles for these datatypes. This technique has been expanded to a whole methodology of structured functional programming, often called origami programming.

In this article, we show how to extend initial algebra semantics from pure inductive datatypes to inductive datatypes interleaved with computational effects. Inductive datatypes interleaved with effects arise naturally in many computational settings. For example, incrementally reading characters from a file generates a list of characters interleaved with input/output actions. Straightforward application of initial algebra techniques to effectful datatypes leads to unnecessarily complicated reasoning, because the pure and effectful concerns must be considered simultaneously. We show how these concerns can be separated the abstraction of initial f-and-m-algebras, where the functor f describes the pure part of a datatype, and the monad m describes the interleaved effects. Because initial f-and-m-algebras are the analogue for the effectful setting of initial f-algebras, they support the extension of the standard definitional and proof principles to the effectful setting. Because initial f-and-m-algebras separate pure and effectful concerns, they support the direct transfer of definitions and proofs from the pure setting to the effectful setting.

Initial f-and-m-algebras are originally due to Filinski and Støvring, and were subsequently generalised to arbitrary categories by the authors of this article. In this article, we aim to introduce the concept of initial f-and-m-algebras to a general functional programming audience.

1 Introduction

One of the attractions of functional programming is the ease by which programmers may lift the level of abstraction. A central example is the use of higher-order combinators for defining and reasoning about programs that operate on recursively defined datatypes. For example, recursive functions on lists can often by re-expressed in terms of the higher-order function *foldr*, which has the type:

$$\mathit{foldr} :: a \to (e \to a \to a) \to [e] \to a$$

The benefits of expressing recursive functions in terms of combinators like *foldr*, rather than through direct use of recursion, are twofold. Firstly, we are automatically guaranteed

several desirable properties, such as totality (on finite input), without having to do any further reasoning. Secondly, functions defined using *foldr* obey a uniqueness property that allows us to easily derive further properties about them. The style of programming that uses combinators such as *foldr* and its uniqueness property has become known as "origami programming" (Gibbons, 2003), and forms a key part of the general Algebra of Programming methodology (Bird & de Moor, 1997).

Programming and reasoning using higher-order recursion combinators is built upon the category theoretic foundation of initial f-algebras for functors f (Goguen *et al.*, 1978). In initial algebra semantics, datatypes are represented by carriers of initial algebras – i.e. least fixed points of functors – and combinators such as *foldr* are derived from the universal properties of initial algebras. The initial f-algebra methodology has been successful in unifying and clarifying structured functional programming and reasoning on values of recursive datatypes that go far beyond lists and *foldr*.

In this article, we present a class of recursive datatypes where direct use of the initial algebra methodology does *not* provide the right level of abstraction. Specifically, we consider recursive datatypes that interleave pure data with effectful computation. For example, lists of characters that are interleaved with the effectful input computations that read them from some external source can be described by the following datatype declaration:

$$\begin{array}{ll} \textbf{data } \mathit{List}'_{io} & \textbf{newtype } \mathit{List}_{io} = \\ \quad = \mathsf{Nil}_{\mathsf{io}} & \quad \mathsf{List}_{\mathsf{io}}\ (\mathit{IO}\ \mathit{List}'_{io}) \\ \quad \mid \mathsf{Cons}_{\mathsf{io}}\ \mathit{Char}\ \mathit{List}_{io} & \end{array}$$

Using the initial f-algebra methology to program with and reason about such datatypes forces us to mingle the pure and effectful parts of our programs and proofs in a way that obscures their essential properties (as we demonstrate in Section 5). By abstracting out the effectful parts, we arrive at the concept of initial f-and-m-algebras, where f is a functor whose initial algebra that describes the pure part of the datatype, and m is a monad that describes the effects. Initial f-and-m-algebras represent a better level of abstraction for dealing with interleaved data and effects. The key idea is to separate the concerns of dealing with pure data, via f-algebras, and effects, via m-Eilenberg-Moore-algebras. We shall see in Section 7 that this separation has the following benefits:

- *Definitions* of functions on datatypes that interleave data and effects look very similar to their counterparts on pure datatypes. We will use the example of adapting the append function on lists to a datatype of lists interleaved with effects to demonstrate this. The pure part of the computation remains the same, and the effectful part is straightforward. Therefore, definitions of functions on pure datatypes can often be transferred directly to their effectful counterparts.
- *Proofs* about functions on interleaved datatypes also carry over almost unchanged from their pure counterparts. We demonstrate this though the proof of associativity for append on effectful lists, which carries over almost unchanged from the proof of associativity of append for pure lists, except for an additional side condition that is discharged almost trivially.

The concept of initial f-and-m-algebras is originally due to Filinski and Støvring (Filinski & Støvring, 2007), and was subsequently extended to a general category-theoretic

setting for arbitrary functors f by the authors of the current article (Atkey *et al.*, 2012). In this article, we aim to introduce the concept of initial f-and-m-algebras to a general functional programming audience, without the heavy category-theoretic prerequisites of our previous work.

1.1 Interleaving data and effects

To motivate the consideration of interleaved data and effects, we consider a function in the Haskell standard library that is not as straightfoward as its type implies. The *hGetContents* function provides an example of *implicit* interleaving of data with effects. This function has the following type:

$$hGetContents :: Handle \to IO\ [Char]$$

Reading the type of this function, one might assume that it operates by reading all the available data from the file referenced by the given handle as an *IO* action, and then returning the list of characters as pure data. In fact, the standard implementation of this function postpones the input effects until the list is actually accessed by the program. The effect of reading from the file handle is implicitly *interleaved* with any later pure computation on the list. This interleaving is not made apparent in the type of *hGetContents*, and this has the following undesirable consequences:

- Input/output errors that occur during reading (e.g., network failure) are reported by throwing exceptions from pure code, using Haskell's imprecise exceptions facility. Since the actual reading may occur long after the call to *hGetContents* has apparently finished, it can be extremely difficult to determine the scope in which such an exception will be thrown.
- Since it is difficult to predict when the read effects will occur, it is no longer safe for the programmer to close the file handle. The handle is implicitly closed when the end of the file is reached. This means that if the string returned by *hGetContents* is never completely read, the handle will never be closed. Since open file handles are a finite resource shared by all processes on a system, the non-deterministic closing of file handles can be a serious problem with long-running programs.

Despite these flaws, there are good reasons for wishing to interleave the effect of reading with data processing. A primary one is that the file being read may be larger than the available memory, so reading it all into a buffer may not be possible. However, the type of *hGetContents* fails to make the interleaving explicit.

We can make the interleaving explicit by declaring a new type of lists interleaved with some effects described by a monad m. This generalises the definition of $List_{io}$ from Page 2:

$$\begin{array}{ll} \textbf{data}\ List'\ m\ a & \textbf{newtype}\ List\ m\ a = \\ \quad = \mathsf{Nil_m} & \quad \mathsf{List}\ (m\ (List'\ m\ a)) \\ \quad \mid \mathsf{Cons_m}\ a\ (List\ m\ a) & \end{array}$$

A value of type $List\ m\ a$ consists of an effect described by m, then either a $\mathsf{Nil_m}$ to indicate the end of the list, or a $\mathsf{Cons_m}$ with a value of type a and another value of type $List\ m\ a$. Thus this datatype describes lists of values of type a interleaved with effects from the monad m.

We can re-implement *hGetContents* to generate values of type *List IO Char*. An simple implementation can be given in terms of the standard Haskell primitives for performing IO on file handles:

$$\begin{aligned}
&\mathit{hGetContents} :: \mathit{Handle} \to \mathit{List\ IO\ Char} \\
&\mathit{hGetContents}\ h = \mathsf{List}\ (\mathbf{do}\ \mathit{isEOF} \leftarrow \mathit{hIsEOF}\ h \\
&\qquad\qquad\qquad\qquad \mathbf{if}\ \mathit{isEOF}\ \mathbf{then}\ \mathit{return}\ \mathsf{Nil} \\
&\qquad\qquad\qquad\qquad \mathbf{else\ do}\ c \leftarrow \mathit{hGetChar}\ h \\
&\qquad\qquad\qquad\qquad\qquad \mathit{return}\ (\mathsf{Cons}\ c\ (\mathit{hGetContents}\ h)))
\end{aligned}$$

By using the *List IO Char* datatype, we have made the possibility of effects between the elements of the list explicit. Therefore, the problems we identified above with implicit interleaving are solved: input/output failures are reported within the scope of *IO* actions, and we have access to the *IO* monad to explicitly close the file.

The Haskell community has also defined many other datatypes that capture the interleaving of effects with pure data[1], in order to make explicit the interleaving implicit in the *hGetContents* function. One of the earliest was Kiselyov's Iteratees, later described in a conference publication (Kiselyov, 2012). In essence, Iteratees are descriptions of functions that alternate reading from some input with effects in some monad, eventually yielding some output. This can be described by the following datatype, which follows the same pattern of mutual recursion as the *List m a* datatype declaration:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Reader'}\ m\ a\ b \\
&\quad = \mathsf{Input}\ (\mathit{Maybe}\ a \to \mathit{Reader}\ m\ a\ b) \\
&\quad \mid \mathsf{Yield}\ b
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathbf{newtype}\ \mathit{Reader}\ m\ a\ b = \\
&\quad \mathsf{Reader}\ (m\ (\mathit{Reader'}\ m\ a\ b))
\end{aligned}$$

A value of type *Reader m a b* is some effect described by the monad *m*, yielding either a result of type *b*, or a request for input. As Kiselyov demonstrates, the fact that values of type *Reader m a b* abstract the source of the data that they read is extremely powerful: different constructions allow values of type *Reader m a b* to be chained together, or connected to actual input/output devices, all while retaining the ability to perform concrete effects in the monad *m*. Kiselyov treats the *Reader m a b* type in isolation. In this article, we show that it can be regarded as an instance of an application of interleaved data and effects: the general construction of the coproduct of a free monad and an arbitrary monad. Monad coproducts provide a general and canonical way of specifying the combination of two monads (Lüth & Ghani, 2002). The construction of the coproduct of a free monad and an arbitrary monad builds on the general foundation of initial f-and-m-algebras.

1.2 The content of this article

We aim to make this article relatively self-contained, so we include the necessary background to enable the reader to follow our proofs and definitions. The structure of the remainder of the article is as follows:

[1] For example, the `iteratees`, `iterIO`, `conduits`, `enumerators`, and `pipes` Haskell libraries all make use of interleaved data and effects. These libraries are all available from the Hackage archive of Haskell libraries (`http://hackage.haskell.org/packages/hackage.html`).

- In Section 2, we recall the standard definitions of f-algebra and initial f-algebras, in a functional programming context. We highlight the proof principle associated with initial f-algebras (Proof Principle 1), and demonstrate that the initial f-algebra abstraction can be thought of as an abstract interface for programming and reasoning.
- We introduce our main running example of list append and its associativity property in Section 3. In this section, we make use of the initial f-algebra methodology for pure datatypes to define list append, and also to show how Proof Principle 1 is used to prove its associativity property.
- In Section 4, we recall the definition of monads, and briefly discuss the work of Fokkinga (Fokkinga, 1994) and Pardo (Pardo, 2004) on recursion schemes in the presence of effects. As we note there, their work covers the case of effectful computations on pure data, whereas we wish to consider computations on effectful data.
- To motivate the use of f-and-m-algebras, in Section 5 we attempt to define and prove associative the append function for effectful lists directly from Proof Principle 1. This turns out to be unnecessarily complicated and loses the direct simplicity of the proof in the pure case.
- In Section 6, we present the definition of f-and-m-algebras, and highlight the associated proof principle (Proof Principle 2). Initial f-and-m-algebras raise our level of abstraction by separating the concerns of pure data and effectful computation. We demonstrate the usefulness of this separation in Section 7, where we revisit the definition of list append on effectful lists, and its associativity property. Using initial algebra semantics for f-and-m-algebras, we are able to reuse much of the definition and proof from the pure case in Section 3, and the additional work that we need to carry out to deal with effects is minimal.
- In Section 8, we show that the construction of initial f-and-m-algebras can be reduced to initial $(f \circ m)$-algebras. Consequently, we are able to give a generic construction of initial f-and-m-algebras for arbirary functors f and monads m.
- Finally, in Section 9, we present an extended example of the use of initial f-and-m-algebras. We reconstruct a result of Hyland, Plotkin and Power on the construction of coproducts of free monads with arbitrary monads, using f-and-m-algebras. This construction has practical interest, in that it provides a general construction of monads that interleave abstract and concrete commands, as seen in the *Reader m a b* monad in the previous section. The monad coproduct structure further highlights the central theme of this article: the separation of concerns of pure data and effects.

2 Background: initial f-algebras

Initial f-and-m-algebras build upon the basic foundation of initial f-algebras. We recall the definition of initial f-algebras in this section, and derive the accompanying definitional and proof principles. We will make use of the basic definitions of the polymorphic identity function $id = \lambda x.\ x$ and function composition $g \circ h = \lambda x.\ g\ (h\ x)$.

2.1 Basic definitions

The initial f-algebra methodology uses functors f to describe the individual "layers" of recursive datatypes. Formally, functors are defined as follows:

Definition 1
A *functor* is a pair $(f, fmap_f)$ of a type operator f and a function $fmap_f$ of type:

$$fmap_f :: (a \to b) \to f\ a \to f\ b$$

such that $fmap_f$ preserves the identity function and composition:

$$fmap_f\ id = id \tag{1}$$

$$fmap_f\ (g \circ h) = fmap_f\ g \circ fmap_f\ h \tag{2}$$

In Haskell, the fact that a type operator f has an associated $fmap_f$ is usually expressed by declaring that f is a member of the *Functor* typeclass:

$$\textbf{class}\ Functor\ f\ \textbf{where}$$
$$fmap :: (a \to b) \to f\ a \to f\ b$$

It is left to the programmer to verify that the identity and composition laws are satisfied. The use of typeclasses to represent functors allows the programmer to just write *fmap* and let the type checker infer which f's associated *fmap* was intended. However, in the interest of clarity, we shall always use a subscript on *fmap* to indicate which type operator is intended.

An f-algebra for a given functor f is a way of describing an operation for reducing each layer in an inductive data structure to a value. Formally, f-algebras are defined as follows:

Definition 2
An *f-algebra* is a pair $(a, fAlgebra_a)$ of a *carrier type* a and a *structure map* $fAlgebra_a :: f\ a \to a$.

Given a pair of f-algebras, there is also the concept of a homomorphism between them:

Definition 3
Given a pair of f-algebras $(a, fAlgebra_a)$ and $(b, fAlgebra_b)$, an *f-algebra homomorphism* between them is a function $h :: a \to b$ such that

$$h \circ fAlgebra_a = fAlgebra_b \circ fmap_f\ h \tag{3}$$

Definition 4
An *initial f-algebra* is an f-algebra $(\mu f, construct)$ such that for any f-algebra $(a, fAlgebra_a)$, there exists a unique f-algebra homomorphism $(\!|fAlgebra_a|\!) :: \mu f \to a$.

The requirement that an initial f-algebra always has an f-algebra homomorphism to any f-algebra allows us to define functions on the datatypes represented by carriers μf of initial f-algebras. The uniqueness requirement yields the following proof principle for functions defined on initial f-algebras.

Proof Principle 1 (*Initial f-Algebras*)
Suppose that $(\mu f, construct)$ is an initial f-algebra.

Let $(a, fAlgebra)$ be an f-algebra, and $g :: \mu\ f \to a$ be a function. To prove an equation

$$(\!|fAlgebra|\!) = g,$$

it suffices to show that g is an f-algebra homomorphism:

$$g \circ construct = fAlgebra \circ fmap_f\ g.$$

We demonstrate the use of Proof Principle 1 in Section 3 below, to set up our presentation of f-and-m-algebras and their associated proof principle. Jacobs and Rutten (Jacobs & Rutten, 2011) further develop the use of Proof Principle 1 (and its dual notion for final coalgebras) for reasoning about recursive programs on pure data.

2.2 Examples of initial f-algebras

The usefulness of the initial f-algebra abstraction for functional programming lies in the fact that we can directly implement initial f-algebras in functional programming languages. We give two examples of implementations of initial f-algebras. The first example shows that standard recursively defined Haskell datatypes can be retrofitted with the initial f-algebra structure. The second example shows that it is possible, in Haskell, to construct an initial f-algebra for any functor $(f, fmap_f)$.

Example 1
The functor *ListF a* describes the individual layers of a list:

$$\begin{array}{ll} \textbf{data}\ \mathit{ListF}\ a\ x & \mathit{fmap}_{\mathit{ListF}\ a} :: (x \to y) \to \mathit{ListF}\ a\ x \to \mathit{ListF}\ a\ y \\ \quad = \mathsf{Nil} & \mathit{fmap}_{\mathit{ListF}\ a}\ g\ \mathsf{Nil} \quad\quad = \mathsf{Nil} \\ \quad \mid \mathsf{Cons}\ a\ x & \mathit{fmap}_{\mathit{ListF}\ a}\ g\ (\mathsf{Cons}\ a\ x) = \mathsf{Cons}\ a\ (g\ x) \end{array}$$

The following definitions witness that the standard Haskell list datatype $[a]$ is the carrier of an initial *ListF a* algebra:

$$\begin{array}{l} \mathit{construct} :: \mathit{ListF}\ a\ [a] \to [a] \\ \mathit{construct}\ \mathsf{Nil} \quad\quad = [\,] \\ \mathit{construct}\ (\mathsf{Cons}\ a\ xs) = a : xs \end{array}$$

and

$$\begin{array}{l} (\!|-|\!) :: (\mathit{ListF}\ a\ b \to b) \to [a] \to b \\ (\!|\mathit{fAlgebra}|\!)\ [\,] \quad\quad = \mathit{fAlgebra}\ \mathsf{Nil} \\ (\!|\mathit{fAlgebra}|\!)\ (a : xs) = \mathit{fAlgebra}\ (\mathsf{Cons}\ a\ ((\!|\mathit{fAlgebra}|\!)\ xs)) \end{array}$$

Example 2
We can implement the carrier of an initial f-algebra for an arbitrary functor $(f, fmap_f)$ as a recursive datatype:

$$\textbf{data}\ \mathit{Mu}\ f = \mathsf{In}\ \{\mathit{unIn} :: f\ (\mathit{Mu}\ f)\}$$

We have used Haskell's record definition syntax to implicitly define a function $\mathit{unIn} :: \mathit{Mu}\ f \to f\ (\mathit{Mu}\ f)$ that is the inverse of the value constructor In. The f-algebra structure map is defined as the value constructor In:

$$\begin{array}{l} \mathit{construct} :: f\ (\mathit{Mu}\ f) \to \mathit{Mu}\ f \\ \mathit{construct} = \mathsf{In} \end{array}$$

and the f-algebra homomorphisms out of $\mathit{Mu}\ f$ are defined in terms of the functor structure fmap_f and Haskell's general recursion:

$$\begin{array}{l} (\!|-|\!) :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to \mathit{Mu}\ f \to a \\ (\!|\mathit{fAlgebra}|\!) = \mathit{fAlgebra} \circ \mathit{fmap}_f\ (\!|\mathit{fAlgebra}|\!) \circ \mathit{unIn} \end{array}$$

This construction has been called "two-level types" (Sheard & Pasalic, 2004), due to the separation between the functor f and the recursive datatype *Mu*.

These two examples demonstrate that initial algebras for a given functor are not unique: the types $[a]$ and $Mu\ (ListF\ a)$ are not equal, but they are both initial $(ListF\ a)$-algebras. Therefore, we regard the initial f-algebra abstraction as an interface to program against, rather than thinking in terms of specific implementations such as $Mu\ f$. Note that it is possible to prove that any two initial f-algebras are isomorphic, by using the initial algebra property to define the translations between, and Proof Principle 1 to prove that the translations are mutually inverse. This isomorphism result is known as Lambek's lemma (Lambek, 1968).

3 List append I: pure lists

We now introduce our running example of list append and its associativity property. In this section, we use an initial $(ListF\ a)$-algebra and Proof Principle 1 to define and prove associative the append function on pure lists. In Section 5 we attempt the same example in a setting with interleaved effects, using the initial f-algebra technique, and see that direct use of initial f-algebras makes the definition and proof unnecessarily complicated. In Section 6, we use f-and-m-algebras to simplify the definition and proof, and show that this lets us reuse much of the definition and proof that we give in this section.

The definition and proof that we present here are standard and have appeared many times in the literature. We present them in some detail in order to use them as a reference when we cover the analogous proof for append for lists interleaved with effects.

We program and reason against the abstract interface of initial algebras. Hence we assume that an initial $(ListF\ a)$-algebra $(\mu(ListF\ a), construct)$ exists, and we write $(\!|-|\!)$ for the unique homomorphism induced by initiality. We can define *append* in terms of $(\!|-|\!)$ as:

$$
\begin{array}{l}
append :: \mu(ListF\ a) \to \mu(ListF\ a) \to \mu(ListF\ a) \\
append\ xs\ ys = (\!|fAlgebra|\!)\ xs \\
\quad \textbf{where}\ fAlgebra :: ListF\ a\ (\mu(ListF\ a)) \to \mu(ListF\ a) \\
\qquad\qquad fAlgebra\ \mathsf{Nil} = ys \\
\qquad\qquad fAlgebra\ (\mathsf{Cons}\ a\ xs) = construct\ (\mathsf{Cons}\ a\ xs)
\end{array}
$$

Immediately from the definition of *append* we know that it is a $(ListF\ a)$-algebra homomorphism in its first argument because it is defined in terms of $(\!|-|\!)$. Unfolding the definitions shows that the following two equational properties of *append* hold. These tell us how it operates on lists of the form $construct\ \mathsf{Nil}$ and $construct\ (\mathsf{Cons}\ a\ xs)$. We have:

$$append\ (construct\ \mathsf{Nil})\ ys = ys \tag{4}$$

$$append\ (construct\ (\mathsf{Cons}\ a\ xs))\ ys = construct\ (\mathsf{Cons}\ a\ (append\ xs\ ys)) \tag{5}$$

We now make use of these properties, and Proof Principle 1, to prove associativity:

Theorem 1
For all $xs, ys, zs :: \mu(ListF\ a)$,

$$append\ xs\ (append\ ys\ zs) = append\ (append\ xs\ ys)\ zs$$

Proof
The function *append* is defined in terms of the initial algebra property of $\mu(ListF\ a)$, so we can use Proof Principle 1 to prove the equation:

$$(\!|fAlgebra|\!)\ xs = append\ (append\ xs\ ys)\ zs$$

In this instantiation of Proof Principle 1, $g = \lambda xs.\ append\ (append\ xs\ ys)\ zs$, and:

$$fAlgebra\ \mathsf{Nil} = append\ ys\ zs \tag{6}$$
$$fAlgebra\ (\mathsf{Cons}\ a\ xs) = construct\ (\mathsf{Cons}\ a\ xs) \tag{7}$$

Thus we need to prove that for all $x :: ListF\ a\ (\mu(ListF\ a))$,

$$\begin{array}{ll} & append\ (append\ (construct\ x)\ ys)\ zs \\ = & fAlgebra\ (fmap_{ListF\ a}\ (\lambda xs.\ append\ (append\ xs\ ys)\ zs)\ x) \end{array}$$

There are two cases to consider, depending on whether $x = \mathsf{Nil}$ or $x = \mathsf{Cons}\ a\ xs$. In the first case, we reason as follows:

$$\begin{array}{ll} & append\ (append\ (construct\ \mathsf{Nil})\ ys)\ zs \\ = & \qquad \{\text{Equation 4}\} \\ & append\ ys\ zs \\ = & \qquad \{\text{definition of } fAlgebra \text{ (Equation 6)}\} \\ & fAlgebra\ \mathsf{Nil} \\ = & \qquad \{\text{definition of } fmap_{ListF\ a}\} \\ & fAlgebra\ (fmap_{ListF\ a}\ (\lambda xs.\ append\ (append\ xs\ ys)\ zs)\ \mathsf{Nil}) \end{array}$$

The other possibility is that $x = \mathsf{Cons}\ a\ xs$, and we reason as follows:

$$\begin{array}{ll} & append\ (append\ (construct\ (\mathsf{Cons}\ a\ xs))\ ys)\ zs \\ = & \qquad \{\text{Equation 5}\} \\ & append\ (construct\ (\mathsf{Cons}\ a\ (append\ xs\ ys)))\ zs \\ = & \qquad \{\text{Equation 5}\} \\ & construct\ (\mathsf{Cons}\ a\ (append\ (append\ xs\ ys)\ zs)) \\ = & \qquad \{\text{definition of } fAlgebra \text{ (Equation 7)}\} \\ & fAlgebra\ (\mathsf{Cons}\ a\ (append\ (append\ xs\ ys)\ zs)) \\ = & \qquad \{\text{definition of } fmap_{ListF\ a}\} \\ & fAlgebra\ (fmap_{ListF\ a}\ (\lambda xs.\ append\ (append\ xs\ ys)\ zs)\ (\mathsf{Cons}\ a\ xs)) \quad \square \end{array}$$

Thus the proof that *append* is associative is relatively straightforward, using Proof Principle 1. We shall see below, in Section 5, that attempting to use Proof Principle 1 again to reason about lists interleaved with effects leads to a more complicated proof that mingles the reasoning above with reasoning about monadic effects. We then make use of f-and-m-algebras in Section 6 to prove the same property for lists interleaved with effects, and show that we are able to reuse the core of the above proof.

4 Background: monadic effects and monadic recursion schemes

As is standard in Haskell programming, we describe effectful computations in terms of monads (Moggi, 1991; Peyton Jones & Wadler, 1993). In order to keep this article self

contained, we recall the definition of monad here. We also briefly examine prior work by Fokkinga, and by Pardo, on effectful recursive programs on pure data.

4.1 Monadic Effects

We have opted to use the "categorical" definition of monad in terms of a *join* (or *multiplication*) operation, rather than the Kleisli-triple presentation with a bind operation ($\gg=$) that is more standard in Haskell programming. For our purposes, the categorical definition is more convenient for equational reasoning. Standard references such as the lecture notes by Benton, Hughes and Moggi (Benton *et al.*, 2000) discuss the translations between the two presentations.

Definition 5
A monad is a quadruple $(m, fmap_m, return_m, join_m)$ of a type constructor m, and three functions:

$$\begin{aligned} fmap_m &:: (a \to b) \to m\ a \to m\ b \\ return_m &:: a \to m\ a \\ join_m &:: m\ (m\ a) \to m\ a \end{aligned}$$

such that the pair $(m, fmap_m)$ is a functor (Definition 1), and the following properties are satisfied:

$$join_m \circ return_m = id \tag{8}$$

$$join_m \circ fmap_m\ return_m = id \tag{9}$$

$$join_m \circ fmap_m\ join_m = join_m \circ join_m \tag{10}$$

and also the naturality laws:

$$return_m \circ f = fmap_m\ f \circ return_m \tag{11}$$

$$join_m \circ fmap_m\ (fmap_m\ f) = fmap_m\ f \circ join_m \tag{12}$$

As with functors and the *Functor* typeclass, monads in Haskell are usually represented in terms of the *Monad* typeclass. Again, for this article, we will always use subscripts on $return_m$ and $join_m$ to disambiguate which monad is being referred to, instead of leaving it for the reader to infer.

Finally in this short recap of monads, we recall the definition of a *monad morphism* between a pair of monads. Monad morphisms represent structure preserving maps between monads. We will use monad morphisms in our extended example of the use of f-and-m-algebras to construct the coproduct of two monads in Section 9.

Definition 6
Let $(m_1, fmap_{m_1}, return_{m_1}, join_{m_1})$ and $(m_2, fmap_{m_1}, return_{m_2}, join_{m_2})$ be a pair of monads. A *monad morphism* between them is a function $h :: m_1\ a \to m_2\ a$ such that:

$$h \circ fmap_{m_1}\ g = fmap_{m_2}\ g \circ h \tag{13}$$

$$h \circ return_{m_1} = return_{m_2} \tag{14}$$

$$h \circ join_{m_1} = join_{m_2} \circ h \circ fmap_{m_1}\ h \tag{15}$$

4.2 Monadic Recursion Schemes

The initial f-algebra methodology has been extended to effectful computation on pure data by Fokkinga (Fokkinga, 1994) and Pardo (Pardo, 2004). The generic recursion combinator they use for effectful recursive computations has the following type:

$$(\!|-|\!)_m : (f\ a \to m\ a) \to \mu f \to m\ a$$

where they also make the implicit assumption that there is a *distributive law* $d :: f\ (m\ a) \to m\ (f\ a)$ that describes how effects percolate through pure data. The assumption that a distributive law exists is too strong for our purposes. In the theories of Fokkinga and Pardo, the distributive law is used to push effects through the pure data in a uniform way. Therefore, they treat the case of effectful structural recursion over *pure* data, where all the effects are pushed to the "outside". By contrast, we wish to deal with structural recursion over *effectful* data, where data and effects are interleaved.

5 List append II: lists with interleaved effects, via f-algebras

Given the success of initial f-algebras for defining and reasoning about programs that operate on pure datatypes, it seems reasonable that they might extend to programming and reasoning about programs that operate on effectful datatypes like *List m a* and *Reader m a b*. As we shall see, it is possible to use initial f-algebras for reasoning about programs on effectful datatypes, but the proofs become unnecessarily complicated. We demonstrate this through an extension of the list append example from Section 3 to the case of lists with interleaved effects.

Our presentation is parameteric in the kind of effects that are interleaved with the list. We merely assume that they can be described by some monad $(m, fmap_m, return_m, join_m)$.

By inspecting the auxillary declarations of *List′ m a* and *Reader′ m a b*, and comparing them to the examples of initial f-algebras that we presented in the Section 2, we can see that they are themselves carriers of initial $(f \circ m)$-algebras, where f is an appropriate functor and $\circ$ denotes functor composition. For example, *List m a* is isomorphic to $m\ (\mu(ListF\ a \circ m))$, where $\mu(ListF\ a \circ m)$ is the carrier of some initial $(ListF\ a \circ m)$-algebra.

Equipped with this observation, we can proceed with adapting the definition of *append* that we gave in Section 3 to the setting of lists interleaved with effects. As above, we program and reason against the abstract interface of initial algebras. We assume that an initial $(ListF\ a \circ m)$-algebra $(\mu(ListF\ a \circ m), construct)$ exists, and we write $(\!|-|\!)$ for the unique homomorphism induced by initiality. We now define *eAppend* ("*e*" for effectful) by:

$$\begin{aligned}
&eAppend :: m\ (\mu(ListF\ a \circ m)) \to m\ (\mu(ListF\ a \circ m)) \to m\ (\mu(ListF\ a \circ m)) \\
&eAppend\ xs\ ys = join_m\ (fmap_m\ (\!|fAlgebra|\!)\ xs) \\
&\quad \textbf{where}\ fAlgebra\ \mathsf{Nil} \quad\quad\quad = ys \\
&\qquad\quad fAlgebra\ (\mathsf{Cons}\ a\ xs) = return_m\ (construct\ (\mathsf{Cons}\ a\ (join_m\ xs)))
\end{aligned}$$

This definition bears a slight resemblance to the definition of *append* above, but we have had to insert uses of the monadic structure $return_m$, $join_m$ and $fmap_m$ to deal with the management of effects. Thus we have had to intermingle the effectful parts of the definition

with the pure parts. This is a result of the fact that the initial f-algebra abstraction is unaware of the presence of effects.

As we did for *append* in Equation 4 and Equation 5 above, we can derive two properties of *eAppend* that tell us how it acts on the two list constructors:

$$eAppend\ (return_m\ (construct\ \mathsf{Nil}))\ ys = ys \tag{16}$$

and

$$\begin{array}{rl} & eAppend\ (return_m\ (construct\ (\mathsf{Cons}\ a\ xs)))\ ys \\ = & return_m\ (construct\ (\mathsf{Cons}\ a\ (eAppend\ xs\ ys))) \end{array} \tag{17}$$

We note that the derivations of these equations involve more work than their counterparts for *append*. In particular, we are forced to spend time shuffling the $return_m$, $join_m$ and $fmap_m$ around in order to apply the monad laws. Evidently, if we were to always use initial f-algebras to define functions on datatypes with interleaved effects, we would be repeating this work over again. Moreover, as we shall see in the proof of Theorem 2 below, we cannot make direct use of Equation 16, because we are forced to unfold the definition of *eAppend* too early.

Theorem 2
For all $xs, ys, zs :: m\ (\mu(ListF\ a \circ m))$,

$$eAppend\ xs\ (eAppend\ ys\ zs) = eAppend\ (eAppend\ xs\ ys)\ zs$$

Proof
We will eventually be able to use Proof Principle 1, but first we must rearrange both sides of the equation to be of a suitable form. For this proof, we adopt the notation $fAlgebra_l$ to denote an instance of the *fAlgebra* function defined in the body of *eAppend* with the free variable *ys* replaced by l.

The left hand side of the equation to be proved is equal to:

$$\begin{array}{rl} & eAppend\ xs\ (eAppend\ ys\ zs) \\ = & \qquad \{\text{definition of } eAppend\} \\ & join_m\ (fmap_m\ (\!|fAlgebra_{eAppend\ ys\ zs}|\!)\ xs) \end{array}$$

The right hand side of the equation requires a little more work:

$$\begin{array}{rl} & eAppend\ (eAppend\ xs\ ys)\ zs \\ = & \qquad \{\text{definition of } eAppend\} \\ & eAppend\ (join_m\ (fmap_m\ (\!|fAlgebra_{ys}|\!)\ xs))\ zs \\ = & \qquad \{\text{definition of } eAppend\} \\ & join_m\ (fmap_m\ (\!|fAlgebra_{zs}|\!)\ (join_m\ (fmap_m\ (\!|fAlgebra_{ys}|\!)\ xs))) \\ = & \qquad \{\text{naturality of } join_m \text{ (Equation 12)}\} \\ & join_m\ (join_m\ (fmap_m\ (fmap_m\ (\!|fAlgebra_{zs}|\!))\ (fmap_m\ (\!|fAlgebra_{ys}|\!)\ xs))) \\ = & \qquad \{\text{monad law: Equation 10}\} \\ & join_m\ (fmap_m\ join_m\ (fmap_m\ (fmap_m\ (\!|fAlgebra_{zs}|\!))\ (fmap_m\ (\!|fAlgebra_{ys}|\!)\ xs))) \\ = & \qquad \{fmap_m \text{ preserves composition (Equation 2)}\} \\ & join_m\ (fmap_m\ (join_m \circ fmap_m\ (\!|fAlgebra_{zs}|\!) \circ (\!|fAlgebra_{ys}|\!))\ xs) \\ = & \qquad \{\text{definition of } eAppend\} \\ & join_m\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ (\!|fAlgebra_{ys}|\!))\ xs) \end{array}$$

Looking at the final lines of these two chains of equations, we see that the problem reduces to proving the following equation:

$$(\!|fAlgebra_{eAppend\ ys\ zs}|\!) = (\lambda l.\ eAppend\ l\ zs) \circ (\!|fAlgebra_{ys}|\!)$$

To prove this equation, we use Proof Principle 1, which reduces the problem to proving the following equation, for all $x :: ListF\ a\ (m\ (\mu(ListF\ a \circ m)))$:

$$\begin{aligned} & eAppend\ ((\!|fAlgebra_{ys}|\!)\ (construct\ x))\ zs \\ = \quad & fAlgebra_{eAppend\ xs\ ys}\ (fmap_{ListF\ a}\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ (\!|fAlgebra_{ys}|\!)))\ x) \end{aligned}$$

There are two cases to consider, depending on whether $x = \mathsf{Nil}$ or $x = \mathsf{Cons}\ a\ xs$. In the first case, we reason as follows. Note that, we are unable to directly apply our knowledge of the effect of *eAppend* on Nil (Equation 16), unlike in the proof of Theorem 1 where we could use Equation 4. This is because we had to unfold the definition of *eAppend* in order to apply Proof Principle 1.

$$\begin{aligned} & eAppend\ ((\!|fAlgebra_{ys}|\!)\ (construct\ \mathsf{Nil}))\ zs \\ = \quad & \{(\!|fAlgebra_{ys}|\!) \text{ is a } (ListF\ a \circ m)\text{-algebra homomorphism}\} \\ & eAppend\ (fAlgebra_{ys}\ (fmap_{ListF\ a}\ (fmap_m\ (\!|fAlgebra_{ys}|\!))\ \mathsf{Nil}))\ zs \\ = \quad & \{\text{definition of } fmap_{ListF\ a}\} \\ & eAppend\ (fAlgebra_{ys}\ \mathsf{Nil})\ zs \\ = \quad & \{\text{definition of } fAlgebra_{ys}\} \\ & eAppend\ ys\ zs \\ = \quad & \{\text{definition of } fAlgebra_{eAppend\ ys\ zs}\} \\ & fAlgebra_{eAppend\ ys\ zs}\ \mathsf{Nil} \\ = \quad & \{\text{definition of } fmap_{ListF\ a}\} \\ & fAlgebra_{eAppend\ xs\ ys}\ (fmap_{ListF\ a}\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ (\!|fAlgebra_{ys}|\!)))\ \mathsf{Nil}) \end{aligned}$$

In the second case, when $x = \mathsf{Cons}\ a\ xs$, we reason using the following steps:

$$\begin{aligned} & eAppend\ ((\!|fAlgebra_{ys}|\!)\ (construct\ (\mathsf{Cons}\ a\ xs)))\ zs \\ = \quad & \{(\!|fAlgebra_{ys}|\!) \text{ is a } (ListF\ a \circ m)\text{-algebra homomorphism}\} \\ & eAppend\ (fAlgebra_{ys}\ (fmap_{ListF\ a}\ (fmap_m\ (\!|fAlgebra_{ys}|\!))\ (\mathsf{Cons}\ a\ xs)))\ zs \\ = \quad & \{\text{definition of } fmap_{ListF\ a}\} \\ & eAppend\ (fAlgebra_{ys}\ (\mathsf{Cons}\ a\ (fmap_m\ (\!|fAlgebra_{ys}|\!)\ xs)))\ zs \\ = \quad & \{\text{definition of } fAlgebra_{ys}\} \\ & eAppend\ (return_m\ (construct\ (\mathsf{Cons}\ a\ (join_m\ (fmap_m\ (\!|fAlgebra_{ys}|\!)\ xs)))))\ zs \\ = \quad & \{\text{definition of } eAppend\} \\ & eAppend\ (return_m\ (construct\ (\mathsf{Cons}\ a\ (eAppend\ xs\ ys))))\ zs \\ = \quad & \{\text{Equation 17}\} \\ & return_m\ (construct\ (\mathsf{Cons}\ a\ (eAppend\ (eAppend\ xs\ ys)\ zs))) \\ = \quad & \{\text{definition of } eAppend\} \\ & return_m\ (construct\ (\mathsf{Cons}\ a \\ & \qquad (join_m\ (fmap_m\ (\!|fAlgebra_{zs}|\!)\ (join_m\ (fmap_m\ (\!|fAlgebra_{ys}|\!)\ xs)))))) \\ = \quad & \{\text{naturality of } join_m \text{ (Equation 12)}\} \\ & return_m\ (construct\ (\mathsf{Cons}\ a \\ & \qquad (join_m\ (join_m\ (fmap_m\ (fmap_m\ (\!|fAlgebra_{zs}|\!))\ (fmap_m\ (\!|fAlgebra_{ys}|\!)\ xs)))))) \end{aligned}$$

$$
\begin{aligned}
&= && \{\text{monad law: } join_m \circ join_m = join_m \circ fmap_m\ join_m \text{ (Equation 10)}\} \\
&\quad return_m\ (construct\ (\mathsf{Cons}\ a \\
&\qquad (join_m\ (fmap_m\ join_m \\
&\qquad\quad (fmap_m\ (fmap_m\ (\!|fAlgebra_{zs}|\!))\ (fmap_m\ (\!|fAlgebra_{ys}|\!)\ xs)))))) \\
&= && \{fmap_m \text{ preserves function composition (Equation 2)}\} \\
&\quad return_m\ (construct\ (\mathsf{Cons}\ a \\
&\qquad (join_m\ (fmap_m\ (join_m \circ fmap_m\ (\!|fAlgebra_{zs}|\!) \circ (\!|fAlgebra_{ys}|\!))\ xs)))) \\
&= && \{\text{definition of } eAppend\} \\
&\quad return_m\ (construct\ (\mathsf{Cons}\ a \\
&\qquad (join_m\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ \circ (\!|fAlgebra_{ys}|\!))\ xs)))) \\
&= && \{\text{definition of } fAlgebra_{eAppend\ ys\ zs}\} \\
&\quad fAlgebra_{eAppend\ ys\ zs}\ (\mathsf{Cons}\ a\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ (\!|fAlgebra_{ys}|\!))\ xs)) \\
&= && \{\text{definition of } fmap_{ListF\ a}\} \\
&\quad fAlgebra_{eAppend\ ys\ zs} \\
&\qquad (fmap_{ListF\ a}\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ (\!|fAlgebra_{ys}|\!)))\ (\mathsf{Cons}\ a\ xs)) \quad \square
\end{aligned}
$$

We identify the following problems with this proof:

- We had to perform a non-trivial number of rewriting steps in order to get ourselves to into a position in which we can apply Proof Principle 1. These steps are not specific to the *eAppend* function, and will have to be re-done whenever we wish to use Proof Principle 1 to prove a property of a function on data interleaved with effects.
- We were forced to unfold the definition of *eAppend* multiple times in order to proceed with the calculation. As we noted during the proof, this unfolding prevented us from applying Equation 16 and instead we had to perform some of the same calculation steps again. For the same reason, in the Cons case, we were only able to apply Equation 17 once, unlike in the proof of Theorem 1 where the analogous equation was applied twice. We also had to expand *eAppend* again in order to rewrite the occurences of $join_m$ and $fmap_m$.

The definition and proof that we have given in this section demonstrates that direct use of initial f-algebras provides us with the wrong level of abstraction for dealing with datatypes that interleave data and effects.

6 Separating data and effects with f-and-m-algebras

As we saw in the previous section, directly defining and proving properties of functions on datatypes consisting of interleaved pure and effectful information is possible, but tedious. We were not able to build upon the definition and proof that we used in the non-effectful case (Section 3), and our equational reasoning repeatedly broke layers of abstraction: we were forced to unfold the definition *eAppend* several times in the proof of Theorem 2 in order to perform further calculation.

To solve the problems we have identified with the direct use of f-algebras, we use the concept of f-and-m-algebras, originally introduced by Filinski and Støvring (Filinski & Støvring, 2007), and generalised to arbitrary functors by the current authors (Atkey *et al.*, 2012). As the name may imply, f-and-m-algebras are simultaneously f-algebras and m-algebras. A twist is that the m-algebra component must be an Eilenberg-Moore algebra.

Eilenberg-Moore algebra structure for a type a describes how to incorporate the effects of the monad m into values of type a.

6.1 Eilenberg-Moore algebras

Given a monad $(m, fmap_m, return_m, join_m)$ (Definition 5), an m-Eilenberg-Moore-algebra is an m-algebra that also interacts well with the structure of the monad:

Definition 7
An *m-Eilenberg-Moore algebra* consists of a pair $(a, mAlgebra_a)$ of a type a and a function

$$mAlgebra_a :: m\ a \to a$$

such that the following two equations are satisfied:

$$mAlgebra_a \circ return_m = id \tag{18}$$

$$mAlgebra_a \circ join_m = mAlgebra_a \circ fmap_m\ mAlgebra_a \tag{19}$$

Eilenberg-Moore algebras form a key piece of the theory of monads, especially in their application to universal algebra. For a monad that represents an algebraic theory (e.g., abelian groups), the collection of all Eilenberg-Moore algebras for that monad are exactly the structures supporting that algebraic theory. Mac Lane's book (Mac Lane, 1998) goes into further depth on this view of Eilenberg-Moore algebras.

In terms of computational effects, an m-Eilenberg-Moore-algebra $(a, mAlgebra)$ represents a way of "performing" the effects of the monad m in the type a, preserving the $return_m$ and $join_m$ of the monad structure. For example, if we let the monad m be the error monad *ErrorM*:

$$\begin{array}{l}\textbf{data}\ ErrorM\ a \\ \quad = \mathsf{Ok}\ a \\ \quad \mid \mathsf{Error}\ String\end{array}$$

$$\begin{array}{l} fmap_{ErrorM}\ g\ (\mathsf{Ok}\ a) = \mathsf{Ok}\ (g\ a) \\ fmap_{ErrorM}\ g\ (\mathsf{Error}\ msg) = \mathsf{Error}\ msg \\ \\ return_{ErrorM}\ a = \mathsf{Ok}\ a \\ \\ join_{ErrorM}\ (\mathsf{Ok}\ (\mathsf{Ok}\ a)) = \mathsf{Ok}\ a \\ join_{ErrorM}\ (\mathsf{Ok}\ (\mathsf{Error}\ msg)) = \mathsf{Error}\ msg \\ join_{ErrorM}\ (\mathsf{Error}\ msg) = \mathsf{Error}\ msg \end{array}$$

then we can define an *ErrorM*-Eilenberg-Moore-algebra with carrier $IO\ a$ as follows:

$$\begin{array}{lcl} mAlgebra :: ErrorM\ (IO\ a) \to IO\ a & & \\ mAlgebra\ (\mathsf{Ok}\ ioa) & = & ioa \\ mAlgebra\ (\mathsf{Error}\ msg) & = & throw\ (\mathsf{ErrorCall}\ msg) \end{array}$$

The function *throw* and the constructor ErrorCall are part of the standard *Control.Exception* module. This *mAlgebra* propagates normal *IO* actions, and interprets errors using the exception throwing facilities of the Haskell *IO* monad.

The general pattern of m-Eilenberg-Moore-algebras with carriers that are themselves constructed from monads has been studied by Filinski under the name "layered monads" (Filinski, 1999). The idea is that the presence of m-Eilenberg-Moore-algebras of the form

$m\ (m'\ a) \to m'\ a$, for all a, capture the fact that the monad m' can perform all the effects that the monad m can, so we can say that m' is layered over m.

A particularly useful class of Eilenberg-Moore algebras for a given monad m is the class of *free* m-Eilenberg-Moore-algebras. The *free* Eilenberg-Moore algebra for an arbitrary type a is given by $(m\ a, join_m)$. In terms of layered monads, this just states that the monad m can be layered over itself. We will make use of this construction below in the proof of Theorem 4 below.

Finally in this short introduction to Eilenberg-Moore algebras, we define homomorphisms between m-Eilenberg-Moore-algebras. These are exactly the same as homomorphisms between f-algebras that we defined in Section 2.

Definition 8
An *m-Eilenberg-Moore-algebra homomorphism*

$$h :: (a, mAlgebra_a) \to (b, mAlgebra_b)$$

consists of a function $h :: a \to b$ such that:

$$h \circ mAlgebra_a = mAlgebra_b \circ fmap_m\ h \tag{20}$$

6.2 Definition of f-and-m-algebras

As we indicated above, an f-and-m-algebra consists of an f-algebra and an m-Eilenberg-Moore-algebra with the same carrier. Intuitively, the f-algebra part deals with the pure parts of the structure, and the m-Eilenberg-Moore-algebra part deals with the effectful parts. We require the extra structure of an Eilenberg-Moore algebra in order to account for the potential merging of the effects that are present between the layers of the inductive datatype (through the preservation of *join*) and the correct preservation of potential lack of effects (through the preservation of *return*).

Definition 9
An *f-and-m-algebra* consists of a triple $(a, fAlgebra, mAlgebra)$ of an object a and two functions:

$$\begin{array}{rcl} fAlgebra & :: & f\ a \to a \\ mAlgebra & :: & m\ a \to a \end{array}$$

where *mAlgebra* is an m-Eilenberg-Moore algebra.

Homomorphisms of f-and-m-algebras are single functions that are simultaneously f-algebra homomorphisms and m-Eilenberg-Moore-algebra homomorphisms:

Definition 10
An *f-and-m-algebra homomorphism*

$$h :: (a, fAlgebra_a, mAlgebra_a) \to (b, fAlgebra_b, mAlgebra_b)$$

between two f-and-m algebras is a function $h :: a \to b$ such that:

$$\begin{array}{rcl} h \circ fAlgebra_a & = & fAlgebra_b \circ fmap_f\ h \\ h \circ mAlgebra_a & = & mAlgebra_b \circ fmap_m\ h \end{array}$$

Given the above definitions, the definition of initial f-and-m-algebras is straightforward, and follows the same structure as for initial f-algebras. Abstractly, an initial f-and-m-algebra is an initial object in the category of f-and-m-algebras and f-and-m-algebra homomorphisms. We use the notation $\mu(f|m)$ for carriers of initial f-and-m-algebras to indicate the interleaving of pure data (represented by f) and effects (represented by m).

Definition 11
An *initial f-and-m-algebra* is an f-and-m-algebra $(\mu(f|m), construct_f, construct_m)$ such that for any f-and-m-algebra $(a, fAlgebra_a, mAlgebra_a)$, there exists a unique f-and-m-algebra homomorphism $(\!|fAlgebra_a|mAlgebra_a|\!) :: \mu(f|m) \to a$.

As for initial f-algebras, the requirement that an initial f-and-m-algebra always has an f-and-m-algebra homomorphism to any other f-and-m-algebra allows us to define functions on the carriers of initial f-and-m-algebras. The uniqueness requirement yields the following proof principle for functions defined on initial f-and-m-algebras. It follows the same basic form as Proof Principle 1 for initial f-algebras, but also includes an obligation to prove that the right-hand side of the equation to be shown is an m-Eilenberg-Moore-algebra homomorphism.

Proof Principle 2 (*Initial f-and-m-Algebras*)
Suppose that $(\mu(f|m), construct_f, construct_m)$ is an initial f-and-m-algebra.

Let $(a, fAlgebra, mAlgebra)$ be some f-and-m-algebra, and let $(\!|fAlgebra|mAlgebra|\!)$ denote the induced function of type $\mu(f|m) \to a$. For any function $g :: \mu(f|m) \to a$, we can prove the equation:

$$(\!|fAlgebra|mAlgebra|\!) = g$$

by demonstrating that

$$g \circ construct_f = fAlgebra \circ fmap_f \tag{21}$$

and

$$g \circ construct_m = mAlgebra \circ fmap_m \tag{22}$$

The key feature of Proof Principle 2 is that it cleanly splits the pure and effectful proof obligations. Therefore we may use this principle to cleanly reason about programs that operate on interleaved pure and effectful data at a high level of abstraction, unlike the direct reasoning we carried out in Section 5. We shall see this separation in action for our list append running example in the next section.

Example 3
The *List m a* datatype that we defined in the introduction can be presented as the carrier of an initial $(ListF\ a)$-and-m-algebra. The $construct_f$ function is defined as follows:

$$\begin{aligned} &construct_f :: ListF\ a\ (List\ m\ a) \to List\ m\ a \\ &construct_f\ \mathsf{Nil} &&= \mathsf{List}\ (return_m\ \mathsf{Nil_m}) \\ &construct_f\ (\mathsf{Cons}\ a\ xs) &&= \mathsf{List}\ (return_m\ (\mathsf{Cons_m}\ a\ xs)) \end{aligned}$$

The $construct_m$ component is slightly complicated by the presence of the List constructor. We use Haskell's **do** notation for convenience:

$$\begin{aligned} &construct_m :: m\ (List\ m\ a) \to List\ m\ a \\ &construct_m\ ml = \mathsf{List}\ (\mathbf{do}\ \{\mathsf{List}\ x \leftarrow ml; x\}) \end{aligned}$$

Finally, we define the induced homomorphism to any other $(ListF\ a)$-and-m-algebra as a pair of mutually recursive functions, following the structure of the declaration of $List\ m\ a$:

$$\begin{array}{l}(\!|-|-|\!) :: (f\ a \to a) \to (m\ a \to a) \to List\ m\ a \to a \\ (\!|fAlgebra|mAlgebra|\!) = loop \\ \quad \textbf{where}\ loop\ (\mathsf{List}\ x) \quad = mAlgebra\ (fmap_m\ loop'\ x) \\ \qquad\quad loop'\ \mathsf{Nil_m} \quad = fAlgebra\ \mathsf{Nil} \\ \qquad\quad loop'\ (\mathsf{Cons_m}\ a\ xs) = fAlgebra\ (\mathsf{Cons}\ a\ (loop\ xs))\end{array}$$

We will give a general construction of initial f-and-m-algebras in Section 8.2 that builds on the generic definition of initial f-algebras from Section 2. The key result is that the existence of initial f-and-m-algebras can be reduced to the existence of initial $(f \circ m)$-algebras: this is Theorem 4 below.

7 List append III: lists with interleaved effects, via f-and-m-algebras

We now revisit the problem of defining and proving associativity for append on lists interleaved with effects that we examined in Section 5. We use the abstraction of (initial) f-and-m-algebras, firstly to simplify the implementation of *eAppend* from Section 5, and secondly to simplify the proof of associativity. We shall see that both the definition and proof mirror the definition and proof from the pure case we presented in Section 3.

By separating the pure and effectful parts of the proof, Proof Principle 2 allows us to reuse proofs from the pure case. Therefore, it makes sense to ask when the additional condition (Equation 21) that it imposes fails. We examine an instance of this in Section 7.2, where a standard property of list reverse fails to carry over to the case of lists with interleaved effects.

7.1 Append for lists with interleaved effects

We define our function *eAppend* against the abstract interface of initial $(ListF\ a)$-and-m-algebras that we defined in the previous section. Hence we assume that an initial $(ListF\ a)$-and-m-algebra $(\mu(ListF\ a|m), construct_{ListF\ a}, construct_m)$ exists, and we denote the unique $(ListF\ a)$-and-m-algebra homomorphism using the notation $(\!|-|-|\!)$. We can define the function *eAppend* in terms of initial f-and-m-algebras as:

$$\begin{array}{l}eAppend :: \mu(ListF\ a|m) \to \mu(ListF\ a|m) \to \mu(ListF\ a|m) \\ eAppend\ xs\ ys = (\!|fAlgebra|construct_m|\!)\ xs \\ \quad \textbf{where}\ fAlgebra\ \mathsf{Nil} \quad = ys \\ \qquad\quad fAlgebra\ (\mathsf{Cons}\ a\ xs) = construct_{ListF\ a}\ (\mathsf{Cons}\ a\ xs)\end{array}$$

Note that, unlike the direct definition of *eAppend* that we made in Section 5, this definition is almost identical to the definition of the function *append* from Section 3. The only differences are the additional m-Eilenberg-Moore-algebra argument to $(\!|-|-|\!)$ and the different type of $construct_{ListF\ a}$. The fact that the pure part of definition (i.e., the function *fAlgebra*) is almost identical to the *fAlgebra* in the definition of *append* is a result of the separation of pure and effectful concerns that the abstraction of f-and-m-algebras affords.

Just as in the case of *append*, we can immediately read off two properties of *eAppend* from the fact that it constructed as a $(ListF\ a)$-algebra homomorphism. We have one

property for each of the constructors of the type constructor *ListF a*:

$$eAppend\ (construct_{ListF\ a}\ \mathsf{Nil})\ ys = ys \tag{23}$$

$$eAppend\ (construct_{ListF\ a}\ (\mathsf{Cons}\ a\ xs))\ ys = construct_{ListF\ a}\ (\mathsf{Cons}\ a\ (eAppend\ xs\ ys)) \tag{24}$$

Again by construction, we also know that *eAppend* is an *m*-Eilenberg-Moore-algebra homomorphism. Hence we have the following property of *eAppend* for free. For all $x :: m\ (\mu(ListF\ a|m))$:

$$eAppend\ (construct_m\ x)\ ys = construct_m\ (fmap_m\ (\lambda xs.\ eAppend\ xs\ ys)\ x) \tag{25}$$

With these three properties of *eAppend* in hand we can prove that it is associative. We use Proof Principle 2, which splits the proof into the pure and effectful parts. As we shall see, the pure part of the proof, where the real work happens, is identical to the proof steps we took in the proof of Theorem 1. The effectful parts of the proof are straightforward, following directly from the fact that *eAppend* is an *m*-Eilenberg-Moore-algebra homomorphism (Equation 25).

Theorem 3

For all $xs, ys, zs :: \mu(ListF\ a|m)$,

$$eAppend\ xs\ (eAppend\ ys\ zs) = eAppend\ (eAppend\ xs\ ys)\ zs$$

Proof

The function *eAppend* is defined in terms of the initial algebra property of $\mu(ListF\ a|m)$, so we can apply Proof Principle 2. Thus we must prove Equation 21 and Equation 22. Firstly, for all $x :: ListF\ a\ (\mu(ListF\ a|m))$, we must show that Equation 21 holds, i.e. that:

$$\begin{array}{rl} & eAppend\ (eAppend\ (construct_{ListF\ a}\ x)\ ys)\ zs \\ = & fAlgebra\ (fmap_{ListF\ a}\ (\lambda xs.\ eAppend\ (eAppend\ xs\ ys)\ zs)\ x) \end{array}$$

where

$$\begin{array}{rcl} fAlgebra\ \mathsf{Nil} & = & eAppend\ ys\ zs \\ fAlgebra\ (\mathsf{Cons}\ a\ xs) & = & construct_{ListF\ a}\ (\mathsf{Cons}\ a\ xs) \end{array}$$

This equation is, up to renaming, *exactly the same* as the equation we had to show in proof of Theorem 1. Therefore, we use the same reasoning steps to show this equation, relying on the properties of *eAppend* captured above in Equation 23 and Equation 24.

Secondly, we must show that the right-hand side of the equation to be proved is an *m*-Eilenberg-Moore-algebra homomorphism, i.e., the Equation 22 holds:

$$\begin{array}{rl} & eAppend\ (eAppend\ (construct_m\ x)\ ys)\ zs \\ = & construct_m\ (fmap_m\ (\lambda xs.\ eAppend\ (eAppend\ xs\ ys)\ zs)\ x) \end{array}$$

This follows straightforwardly from the fact that *eAppend* is itself an *m*-Eilenberg-Moore-algebra homomorphism, as we noted above in Equation 25, and that such homomorphisms

are closed under composition:

$$\begin{aligned}
& \mathit{eAppend}\ (\mathit{eAppend}\ (\mathit{construct}_m\ x)\ ys)\ zs \\
= \quad & \{\text{Equation 25}\} \\
& \mathit{eAppend}\ (\mathit{construct}_m\ (\mathit{fmap}_m\ (\lambda xs.\ \mathit{eAppend}\ xs\ ys)\ x))\ zs \\
= \quad & \{\text{Equation 25}\} \\
& \mathit{construct}_m\ (\mathit{fmap}_m\ (\lambda xs.\ \mathit{eAppend}\ xs\ zs)\ (\mathit{fmap}_m\ (\lambda xs.\ \mathit{eAppend}\ xs\ ys)\ x)) \\
= \quad & \{\mathit{fmap}_m \text{ preserves function composition (Equation 2)}\} \\
& \mathit{construct}_m\ (\mathit{fmap}_m\ (\lambda xs.\ \mathit{eAppend}\ (\mathit{eAppend}\ xs\ ys)\ zs)\ x) \quad \square
\end{aligned}$$

As promised, the proof that *eAppend* is associative, using Proof Principle 2, is much simpler than the direct f-algebra proof we attempted in Section 5. In addition, the separation of pure and effectful parts has meant that we were able to reuse the proof of the pure case from Section 3, and so need only to establish the side condition for effects.

7.2 Reverse for lists with interleaved effects?

Given the above example of a proof of a property of a function on pure lists carrying over almost unchanged to lists interleaved with effects, one might wonder if there are circumstances where this approach fails. Clearly, it cannot be the case that all properties true for pure lists carry over to effectful lists. One example of a property that fails to carry over is the following property of the reverse function:

$$\mathit{reverse}\ (\mathit{append}\ xs\ ys) = \mathit{append}\ (\mathit{reverse}\ ys)\ (\mathit{reverse}\ xs) \tag{26}$$

Intuitively, this property cannot possibly hold for a reverse function on lists interleaved with effects, since in order to reverse a list, all of the effects inside it must be executed in order to reach the last element and place it at the head of the new list. Thus the left hand side of the equation above will execute all the effects of xs and then ys in order, whereas the right hand side will execute all the effects of ys first, and then xs. If we try to prove this property using Proof Principle 2, we see that we are unable to prove Equation 22, namely that the right-hand side must be an Eilenberg-Moore-algebra homomorphism.

We can define a reverse function on effectful lists as follows. This is very similar to the standard definition of (non-tail recursive) reverse on pure lists, and makes use of the *eAppend* function we defined above.

$$\begin{aligned}
& \mathit{eReverse} :: \mu(\mathit{ListF}\ a|m) \to \mu(\mathit{ListF}\ a|m) \\
& \mathit{eReverse} = (\!|\mathit{fAlgebra}|\mathit{construct}_m|\!) \\
& \quad \textbf{where}\ \mathit{fAlgebra}\ \mathsf{Nil} = \mathit{construct}_{\mathit{ListF}\ a}\ \mathsf{Nil} \\
& \qquad\quad \mathit{fAlgebra}\ (\mathsf{Cons}\ a\ xs) = \mathit{eAppend}\ xs\ (\mathit{construct}\ (\mathsf{Cons}\ a\ (\mathit{construct}\ \mathsf{Nil})))
\end{aligned}$$

Verifying the effectful analogue of Equation 26 requires a little extra step before we can apply Proof Principle 2, because the left-hand side of the equation is constructed from a composite of two functions of the form $(\!|-|-|\!)$. However, it is straightforward to prove that this composite is equal to $(\!|\mathit{alg}|\mathit{construct}_m|\!)$, where

$$\begin{aligned}
& \mathit{alg} :: \mathit{ListF}\ a\ (\mu(\mathit{ListF}\ a|m)) \to \mu(\mathit{ListF}\ a|m) \\
& \mathit{alg}\ \mathsf{Nil} = \mathit{eReverse}\ ys \\
& \mathit{alg}\ (\mathsf{Cons}\ a\ xs) = \mathit{eAppend}\ xs\ (\mathit{construct}\ (\mathsf{Cons}\ a\ (\mathit{construct}\ \mathsf{Nil})))
\end{aligned}$$

This same extra step is required in the case for pure datatypes as well, so this is not where the problem with interleaved effects lies. If we attempt to apply Proof Principle 2 to the equation:

$$(\!|alg|construct_m|\!)\ xs = eAppend\ (eReverse\ ys)\ (eReverse\ xs)$$

Then the pure part of the proof goes through straightforwardly. We are left with proving that the right hand side of this equation is an Eilenberg-Moore-algebra homomorphism in its second argument. Certainly, *eReverse* is an Eilenberg-Moore-algebra homomorphism by its construction via the initial f-and-m-algebra property. However, *eAppend* is not an Eilenberg-Moore algebra homomorphism in its *second* argument, as the following counterexample shows. If we let the monad m be the *ErrorM* monad we defined in Section 6.1, then if *eAppend* were an Eilenberg-Moore-algebra homomorphism in its second argument the following equation would hold:

$$\begin{aligned} & eAppend\ (construct_f\ (\mathsf{Cons}\ a\ (construct_f\ \mathsf{Nil})))\ (construct_{ErrorM}\ (\mathsf{Error}\ \texttt{"msg"})) \\ = \ & construct_{ErrorM} \\ & \quad (fmap_{ErrorM}\ (\lambda ys.\ eAppend\ (construct_f\ (\mathsf{Cons}\ a\ (construct_f\ \mathsf{Nil}))))) \\ & \quad\ (\mathsf{Error}\ \texttt{"msg"})) \end{aligned} \tag{27}$$

However, starting from the left-hand side, we calculate as follows:

$$\begin{aligned} & eAppend\ (construct_f\ (\mathsf{Cons}\ a\ (construct_f\ \mathsf{Nil})))\ (construct_{ErrorM}\ (\mathsf{Error}\ \texttt{"msg"})) \\ = \ & \qquad \{\text{Equation 24}\} \\ & construct_f\ (Cons\ a\ (eAppend\ (construct_f\ \mathsf{Nil})\ (construct_{ErrorM}\ (\mathsf{Error}\ \texttt{"msg"})))) \\ = \ & \qquad \{\text{Equation 23}\} \\ & construct_f\ (Cons\ a\ (construct_{ErrorM}\ (\mathsf{Error}\ \texttt{"msg"}))) \end{aligned}$$

while the right hand side of Equation 27 reduces by the definition of $fmap_{ErrorM}$ to simply:

$$construct_{ErrorM}\ (\mathsf{Error}\ \texttt{"msg"})$$

Thus the proof fails. This is the formal rendering of the intuition for the failure given at the start of this subsection.

8 Generic implementation of initial f-and-m-algebras

We have seen that existing datatypes such as *List m a* can be given the structure of initial f-and-m-algebras. In this section, we show that, in Haskell, we can implement an initial f-and-m-algebra for and functor f and monad m. We build on the generic implementation of initial f-algebras we presented in Section 2.2. The key construction is to show that if we have an initial $(f \circ m)$-algebra, then we can construct an initial f-and-m-algebra.

8.1 From initial $(f \circ m)$-algebras to initial f-and-m-algebras

Initial f-and-m-algebras can be constructed from initial $(f \circ m)$-algebras. If the type $\mu(f \circ m)$ is the carrier of an initial $(f \circ m)$-algebra, then the initial f-and-m-algebra that we construct has carrier $m\ (\mu(f \circ m))$. One way of looking at the proof of the following theorem is as containing all the additional parts of the definition and proof steps we carried

out in the direct initial f-algebra proof of associativity in Section 5 that were missing in the initial f-and-m-algebra approach in the previous section. Thus we have abstracted out parts that are common to all definitions and proofs that have to do with interleaved data and effects.

Theorem 4
Let $(f, fmap_f)$ be a functor, and $(m, fmap_m, return_m, join_m)$ be a monad. If we have an initial $(f \circ m)$-algebra $(\mu(f \circ m), construct)$, then $m\ (\mu(f \circ m))$ is the carrier of an initial f-and-m-algebra.

Proof
The f-algebra and m-Eilenberg-Moore-algebra structure are constructed from the $(f \circ m)$-algebra structure map *construct* and the structure of the monad m. For the f-algebra component, we use the composite:

$$construct_f = return_m \circ construct :: f\ (m\ (\mu(f \circ m))) \to m\ (\mu(f \circ m))$$

The m-Eilenberg-Moore-algebra component is straightforward, using the free Eilenberg-Moore-algebra construction from Section 6.1:

$$construct_m = join_m :: m\ (m\ (\mu(f \circ m))) \to m\ (\mu(f \circ m))$$

Since we have used the free Eilenberg-Moore-algebra construction, we are automatically guaranteed that we have an m-Eilenberg-Moore-algebra.

Now let us assume we are given an f-and-m-algebra $(a, fAlgebra_a, mAlgebra_a)$. We construct, and prove unique, an f-and-m-algebra homomorphism h from the algebra $(m\ (\mu(f \circ m)), construct_f, construct_m)$ to the algebra with carrier a using the initiality of $\mu(f \circ m)$:

$$h = mAlgebra_a \circ fmap_m\ (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!) :: m\ (\mu(f \circ m)) \to a$$

Close inspection of h reveals that it has the same structure as the definition of *eAppend* in terms of initial f-algebras we gave in Section 5. Therefore, as we noted in the introduction to this subsection, the construction we are buliding here abstracts out the common parts of proofs and definitions on effectful datatypes.

To complete our proof, we now need to demonstrate that h is an f-and-m-algebra homomorphism, and that it is the unique such. We split this task into three steps:

1. The function h is an f-algebra homomorphism. We reason as follows:

$$\begin{aligned}
& h \circ construct_f \\
= & \quad \{\text{definitions of } h \text{ and } construct_f\} \\
& mAlgebra_a \circ fmap_m\ (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!) \circ return_m \circ construct \\
= & \quad \{\text{naturality of } return_m \text{ (Equation 11)}\} \\
& mAlgebra_a \circ return_m \circ (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!) \circ construct \\
= & \quad \{mAlgebra_a \text{ is an } m\text{-Eilenberg-Moore-algebra (Equation 18)}\} \\
& (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!) \circ construct \\
= & \quad \{(\!|-|\!) \text{ is an } (f \circ m)\text{-algebra homomorphism (Equation 3)}\} \\
& fAlgebra_a \circ \\
& \qquad fmap_f\ mAlgebra_a \circ fmap_f\ (fmap_m\ (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!))
\end{aligned}$$

$$
\begin{array}{ll}
= & \{fmap_f \text{ preserves function composition (Equation 2)}\} \\
& fAlgebra_a \circ fmap_f\ (mAlgebra_a \circ fmap_m\ (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!)) \\
= & \{\text{definition of } h\} \\
& fAlgebra_a \circ fmap_f\ h
\end{array}
$$

2. The function h is an m-Eilenberg-Moore-algebra homomorphism, as shown by the following steps:

$$
\begin{array}{ll}
& h \circ construct_m \\
= & \{\text{definitions of } h \text{ and } construct_m\} \\
& mAlgebra_a \circ fmap_m\ (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!) \circ join_m \\
= & \{\text{naturality of } join_m \text{ (Equation 12)}\} \\
& mAlgebra_a \circ join_m \circ fmap_m\ (fmap_m\ (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!)) \\
= & \{mAlgebra_a \text{ is an Eilenberg-Moore algebra (Equation 19)}\} \\
& mAlgebra_a \circ \\
& \quad fmap_m\ mAlgebra_a \circ fmap_m\ (fmap_m\ (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!)) \\
= & \{fmap_m \text{ preserves function composition (Equation 2)}\} \\
& mAlgebra_a \circ fmap_m\ (mAlgebra_a \circ fmap_m\ (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!)) \\
= & \{\text{definition of } h\} \\
& mAlgebra_a \circ fmap_m\ h
\end{array}
$$

3. The function h is the unique such f-and-m-algebra. Let us assume that there exists another f-and-m-algebra homomorphism $h' :: m\ (\mu(f \circ m)) \to a$. We aim to show that $h = h'$. We first observe that the following function defined by composition:

$$h' \circ return_m :: \mu(f \circ m) \to a$$

is an $(f \circ m)$-algebra homomorphism from $(\mu(f \circ m), construct)$ to $(a, fAlgebra_a \circ fmap_f\ mAlgebra_a)$, as verified by the following steps:

$$
\begin{array}{ll}
& h' \circ return_m \circ construct \\
= & \{\text{definition of } construct_f\} \\
& h' \circ construct_f \\
= & \{h' \text{ is an } f\text{-and-}m\text{-algebra homomorphism}\} \\
& fAlgebra_a \circ fmap_f\ h' \\
= & \{\text{monad law: } join_m \circ fmap_m\ return_m = id \text{ (Equation 9)}\} \\
& fAlgebra_a \circ fmap_f\ (h' \circ join_m \circ fmap_m\ return_m) \\
= & \{h' \text{ is an } m\text{-Eilenberg-Moore-algebra homomorphism (Equation 20)}\} \\
& fAlgebra_a \circ fmap_f\ (mAlgebra_a \circ fmap_m\ h' \circ fmap_m\ return_m) \\
= & \{fmap_f \text{ preserves function composition (Equation 2)}\} \\
& fAlgebra_a \circ fmap_f\ mAlgebra_a \circ fmap_f\ (fmap_m\ (h' \circ return_m))
\end{array}
$$

Thus, by the uniqueness of $(f \circ m)$-algebra homomorphisms out of $\mu(f \circ m)$, we have proved that

$$h' \circ return_m = (\!|fAlgebra_a \circ fmap_f\ mAlgebra_a|\!) \tag{28}$$

We now use this equation to prove that $h = h'$ by the following steps:

$$
\begin{array}{ll}
 & h \\
= & \quad \{\text{definition of } h\} \\
 & mAlgebra_a \circ fmap_m \, (\!|fAlgebra_a \circ fmap_f \; mAlgebra_a|\!) \\
= & \quad \{\text{Equation 28}\} \\
 & mAlgebra_a \circ fmap_m \; (h' \circ return_m) \\
= & \quad \{fmap_m \text{ preserves function composition (Equation 2)}\} \\
 & mAlgebra_a \circ fmap_m \; h' \circ fmap_m \; return_m \\
= & \quad \{h' \text{ is an } m\text{-Eilenberg-Moore-algebra homomorphism (Equation 20)}\} \\
 & h' \circ join_m \circ fmap_m \; return_m \\
= & \quad \{\text{monad law: } join_m \circ fmap_m \; return_m = id\} \\
 & h'
\end{array}
$$

Thus h is the unique f-and-m-algebra homomorphism from $m \; (\mu(f \circ m))$ to a. □

In the current authors' previous work (Atkey *et al.*, 2012), this same result was obtained in a less elementary way by constructing a functor Φ from the category of $(f \circ m)$-algebras to the category of f-and-m-algebras. The functor Φ was shown to be a left adjoint, and since left adjoints preserve initial objects, Φ maps any initial $(f \circ m)$-algebra to an initial f-and-m-algebra.

8.2 Implementation of initial f-and-m-algebras in Haskell

In light of Theorem 4, we can take the Haskell implementation of initial f-algebras from Section 2 and apply the construction in the theorem to construct an initial f-and-m-algebra. We do have to make a small alteration to satisfy Haskell's type checker, however.

The seed of our construction is the existence of an initial $(f \circ m)$-algebra. Therefore, we need to first construct the composite functor $f \circ m$. Since Haskell does not have type-level λ-abstraction or application, there is no lightweight way of constructing the composite of two functors' type operator components. Thus to express the composition of two type operators as a new type operator, we must introduce a **newtype**, as follows[2]:

$$\textbf{newtype} \; (f :\circ: g) \; a = \mathsf{C} \; \{unC :: f \; (g \; a)\}$$

We define $fmap_{f :\circ: g}$ straightforwardly in terms of $fmap_f$ and $fmap_g$:

$$fmap_{f :\circ: g} \; h \; (\mathsf{C} \; x) = \mathsf{C} \; (fmap_f \; (fmap_g \; h) \; x)$$

Theorem 4 states that if $\mu(f \circ m)$ is the carrier of an initial $(f \circ m)$-algebra, then $m \; (\mu(f \circ m))$ is the carrier of an initial f-and-m-algebra. Therefore, we can define an implementation of the initial f-and-m-algebra by setting $\mu(f|m)$ to be the type $MuFM \; f \; m$, which is defined as:

$$\textbf{type} \; MuFM \; f \; m = m \; (Mu \; (f :\circ: m))$$

[2] This definition requires the GHC extension `-XTypeOperators` to be turned on, allowing infix type constructors.

The f-algebra and m-Eilenberg-Moore-algebra structure maps $construct_f$ and $construct_m$ are defined following the construction in Theorem 4, augmented with a use of the value constructor C to satisfy the type checker:

$$\begin{aligned}&construct_f :: f\ (MuFM\ f\ m) \to MuFM\ f\ m\\&construct_f = return_m \circ construct \circ \mathsf{C}\end{aligned}$$

$$\begin{aligned}&construct_m :: m\ (MuFM\ f\ m) \to MuFM\ f\ m\\&construct_m = join_m\end{aligned}$$

Finally, we construct the unique f-and-m-homomorphism out of $MuFM\ f\ m$ following the proof of Theorem 4 by building upon our implementation of the unique homomorphisms out of the initial $(f :\circ: m)$-algebra, albeit augmented with a coercion $unC :: (f :\circ: m)\ a \to f\ (m\ a)$ to satisfy the type checker:

$$\begin{aligned}&(\!|-|-|\!) :: (f\ a \to a) \to (m\ a \to a) \to MuFM\ f\ m \to a\\&(\!|fAlgebra|mAlgebra|\!) = mAlgebra \circ fmap_m\ (\!|fAlgebra \circ fmap_f\ mAlgebra \circ unC|\!)\end{aligned}$$

We can also implement $(\!|-|-|\!)$ directly in terms of Haskell's general recursion, just as we did for the implementation of $(\!|-|\!)$. This definition arises simply by inlining the implementation of $(\!|-|\!)$ into the definition of $(\!|-|-|\!)$ above, and performing some simple rewriting. The direct implementation of $(\!|-|-|\!)$ is as follows:

$$\begin{aligned}&(\!|-|-|\!) :: (f\ a \to a) \to (m\ a \to a) \to MuFM\ f\ m \to a\\&(\!|fAlgebra|mAlgebra|\!) = mAlgebra \circ fmap_m\ loop\\&\quad \textbf{where}\ loop\ =\ fAlgebra \circ fmap_f\ mAlgebra \circ fmap_f\ (fmap_m\ loop) \circ unC \circ unIn\end{aligned}$$

Whichever implementation of $(\!|-|-|\!)$ we choose, we note that there is an implicit precondition that the second argument (of type $m\ a \to a$) must be an Eilenberg-Moore algebra. Unfortunately, we are unable to express this requirement in Haskell's type system.

9 Application: Coproducts of free monads with arbitrary monads

To demonstrate the effectiveness of initial f-and-m-algebras for programming with and reasoning about datatypes with interleaved effects, we reconstruct a result originally due to Hyland, Plotkin and Power (Hyland *et al.*, 2006) on the construction of coproducts of free monads with arbitrary monads. This example highlights the advantages that f-and-m-algebras' separation of pure data and effects provides, and also has theoretical and practical interest. We now give a brief explanation of the relevant concepts.

Monad coproducts provide a canonical way of describing the combination of two monads to form another monad. We formally define the coproduct of two monads below, in Section 9.3. For now, we can think of the coproduct of two monads as the "least commitment" combination. The coproduct of two monads is able to describe any effects that its constituents describes, but imposes no interaction between them. Unfortunately, the coproduct of two arbitrary monads is not always guaranteed to exist, but is known to exist in certain special cases. For example, monad coproducts are guaranteed to exist when the monads in question are ideal monads (Ghani & Uustalu, 2004), or when working in the category of Sets (Adámek *et al.*, 2012), or if the monads are constructed from algebraic

theories (Hyland *et al.*, 2006). One particular special case is when one of the constituent monads is *free*.

A *free monad* for a functor $(f, fmap_f)$ is a way of extending f to be a monad while, intuitively, adding no additional constraints. A useful application of free monads is as a way of describing effectful computations over a set of commands, where the commands are described by the functor f, and no commitment is made as to their interpretations. Swierstra and Altenkirch (Swierstra & Altenkirch, 2007) have developed this idea to provide a straightforward way of reasoning about programs that perform input/output. We briefly describe this view of free monads after we give the formal definition in Section 9.1.

Coproducts of free monads with arbitrary monads arise when considering certain kinds of datatypes interleaved with effects, such as the Iteratee type *Reader m a b* in Section 1.1. This type describes computations that interleave the possibility of reading with performing effects in some arbitrary monad (often the *IO* monad). The power of the *Reader m a b* type, as demonstrated by Kiselyov (Kiselyov, 2012), is that we are provided with considerable flexibility in how to interpret "reading", allowing for instances of the type *Reader m a b* to be chained together in interesting ways. The monad coproduct interface provides us with a simple, general, and canonical way of precisely describing exactly how *Reader m a b* is the combination of read effects with effects in *m*.

In the next subsection, we present the formal definition of the notion of a free monad, and briefly describe the reading of free monads as abstract sequences of commands that can be interpreted in multiple ways. In Section 9.2, we show that concrete free monads can be defined using specific initial f-algebras. We will be able to reuse most of this construction when constructing the coproduct in Section 9.4. We present the formal definition of monad coproduct in Section 9.3, and elaborate on the reading of free monads as sequences of commands, now interleaved with effects from some arbitrary monad. Finally, in Section 9.4, we present a concrete construction of the coproduct of a free monad with an arbitrary monad. By making use of f-and-m-algebras we are able to reuse much of the core of the definitions of the free monad structure we defined in Section 9.2.

We emphasise that the result we present here is not new; Hyland *et al.* have already demonstrated, albeit with a different proof technique, that the construction we give below in Section 9.4 actually defines the monad coproduct. A special case of this result, where the free monad part of the construction is the free monad over the identity functor, has also been previously presented by Piróg and Gibbons (Piróg & Gibbons, 2012). Our contribution is two show that the use of f-and-m-algebras simplifies and elucidates the definitions involved.

9.1 Free monads

In this section, we give the formal definition of the free monad interface, and give a brief example showing how free monads provide a powerful way of writing effectful programs that have multiple interpretations.

Definition 12
Let $(f, fmap_f)$ be a functor. A *free monad* on $(f, fmap_f)$ is a monad

$$(FreeM\ f, fmap_{FreeM\ f}, return_{FreeM\ f}, join_{FreeM\ f})$$

equipped with a function:

$$wrap_f :: f\ (FreeM\ f\ a) \to FreeM\ f\ a$$

that satisfies:

$$wrap_f \circ fmap_f\ (fmap_{FreeM\ f}\ g) = fmap_{FreeM\ f}\ g \circ wrap_f \tag{29}$$

$$wrap_f \circ fmap_f\ join_{FreeM\ f} = join_{FreeM\ f} \circ wrap_f \tag{30}$$

and such that for every monad $(m, fmap_m, return_m, join_m)$ and $g :: f\ a \to m\ a$, such that g is natural:

$$g \circ fmap_f\ k = fmap_m\ k \circ g$$

there is a unique monad morphism $\langle\!\langle g \rangle\!\rangle :: FreeM\ f\ a \to m\ a$ such that:

$$join_m \circ fmap_m\ \langle\!\langle g \rangle\!\rangle \circ g = \langle\!\langle g \rangle\!\rangle \circ wrap_f$$

An alternative but equivalent definition of free monad, which is slightly more standard from a categorical point of view, has the type of $wrap_f$ as $f\ a \to FreeM\ f\ a$. We choose the form in Definition 12 because it is more convenient for programming.

The following lemma is an immediate consequence of the definition of free monad, and can be taken as another alternative definition in terms of isomorphisms of collections of morphisms. It will be useful when we come to define the coproduct of free monads with arbitrary monads in terms of f-and-m-algebras in Section 9.4 below.

Lemma 1
If $(FreeM\ f, fmap_{FreeM\ f}, return_{FreeM\ f}, join_{FreeM\ f})$ is a free monad for a functor $(f, fmap_f)$, then the operation $\langle\!\langle - \rangle\!\rangle :: (\forall a.\ f\ a \to m\ a) \to (\forall a.\ FreeM\ f\ a \to m\ a)$ is a bijection between natural transformations and monad morphisms. The inverse operation can be defined as follows:

$$\langle\!\langle - \rangle\!\rangle^{-1} :: (\forall a.\ FreeM\ f\ a \to m\ a) \to (\forall a.\ f\ a \to m\ a)$$
$$\langle\!\langle h \rangle\!\rangle^{-1} = h \circ wrap_f \circ fmap_f\ return_{FreeM\ f}$$

One way of explaining the free monad abstraction is in terms of expressions with variables, and substitution. Under this reading, the functor $(f, fmap_f)$ describes the constructors that can be used to make expressions, and a value of type $FreeM\ f\ a$ is an expression comprised of the constructors from f and variables from a. The $join_{FreeM\ f}$ part of the monad structure provides substitution of expressions into other expressions, and the extension $\langle\!\langle g \rangle\!\rangle$ allows us to interpret a whole expression if we can interpret all the constructors.

Another reading, which is more in line with our general theme of computational effects, is in terms of sequences of "commands". We think of the functor $(f, fmap_f)$ as describing a collection of possible commands that can be issued by a program. For example, the functor $(ReaderF\ a, fmap_{ReaderF\ a})$, that we define now, describes a single command of reading a value from some input. The $ReaderF\ a$ functor is defined as follows:

$$\begin{array}{ll} \textbf{data}\ ReaderF\ a\ x & fmap_{ReaderF\ a} :: (x \to y) \to ReaderF\ a\ x \to ReaderF\ a\ y \\ \quad = \mathsf{Read}\ (a \to x) & fmap_{ReaderF\ a}\ g\ (\mathsf{Read}\ k) = \mathsf{Read}\ (g \circ k) \end{array}$$

We think of values of type $FreeM\ (ReaderF\ a)\ b$ as sequences of read commands, eventually yielding a value of type b. We use the $wrap_{ReaderF\ a}$ part of the free monad interface to

define a primitive read operation:

$$\begin{aligned} &\mathit{read} :: \mathit{FreeM}\ (\mathit{ReaderF}\ a)\ a \\ &\mathit{read} = \mathit{wrap}_{\mathit{ReaderF}\ a}\ (\mathsf{Read}\ \mathit{return}_{\mathit{FreeM}\ (\mathit{ReaderF}\ a)}) \end{aligned}$$

As every free monad is a monad, we can use Haskell's **do** notation to sequence individual commands. For example, here is a simple program that reads two strings from some input, and returns them as a pair in the opposite order.

$$\begin{aligned} &\mathit{swapRead} :: \mathit{FreeM}\ (\mathit{ReaderF}\ \mathit{String})\ (\mathit{String}, \mathit{String}) \\ &\mathit{swapRead} = \mathbf{do}\ \{s_1 \leftarrow \mathit{read}; s_2 \leftarrow \mathit{read}; \mathit{return}\ (s_2, s_1)\} \end{aligned}$$

The free monad interface gives us considerable flexibility in how we actually interpret the *read* commands. For example, we can interpret each *read* command as reading a line from the terminal by defining a transformation from *ReaderF String* to *IO*, using the standard Haskell function *getLine* to do the actual reading:

$$\begin{aligned} &\mathit{useGetLine} :: \mathit{ReaderF}\ \mathit{String}\ a \to \mathit{IO}\ a \\ &\mathit{useGetLine}\ (\mathsf{Read}\ k) = \mathbf{do}\ \{s \leftarrow \mathit{getLine}; \mathit{return}\ (k\ s)\} \end{aligned}$$

The free monad interface now provides a way to extend this interpretation of individual commands to sequences of commmands:

$$\langle\!\langle \mathit{useGetLine} \rangle\!\rangle :: \mathit{FreeM}\ (\mathit{ReaderF}\ \mathit{String})\ a \to \mathit{IO}\ a$$

Applying $\langle\!\langle \mathit{useGetLine} \rangle\!\rangle$ to *swapRead* results in the following interaction, where the second and third lines are entered by the user, and the final line is printed by the Haskell implementation:

```
> ⟪useGetLine⟫ swapRead
"free"
"monad"
("monad","free")
```

The free monad interface provides us with a powerful way of giving multiple interpretations to effectful commands. Moreover, it is easy to extend the language of commands simply by extending the functor f. Swierstra (Swierstra, 2008) demonstrates a convenient method in Haskell for dealing with modular construction of functors for describing commands in free monads. However, explicitly naming every additional command that we wish to be able to perform can be tedious. Sometimes, we simply want access to effects in a known monad m. For example, we may know that we want to execute concrete *IO* actions as well as abstract read operations. One possible way of accomplishing this is to ensure that there is an additional constructor to the functor f that describes an additional "abstract command" of performing an effect in the chosen monad. For example, we could extend the *ReaderF a* functor like so to add the possibility of concrete effects in a monad m:

$$\mathbf{data}\ \mathit{ReaderMF}\ m\ a\ x = \mathsf{Read}\ (a \to x) \mid \mathsf{Act}\ (m\ x)$$

This approach has the disadvantage that the effects of the monad m must now be handled by the interpretation of the other abstract commands. For example, we would have to add another case to the *useGetLine* function to handle the Act case. Thus, we would be forced to combine the interpretation of the pure data representing abstract commands with

Let $(f, fmap_f)$ be a functor, and define:

```
data FreeMF f a x
  =  Var a
  |  Term (f x)

fmap_FreeMF :: (x → y) → FreeMF f a x → FreeMF f a y
fmap_FreeMF g (Var a)    = Var a
fmap_FreeMF g (Term fx) = Term (fmap_f g fx)
```

Free monads:

```
type FreeM f a = μ(FreeMF f a)

fmap_FreeM f :: (a → b) → FreeM f a → FreeM f b
fmap_FreeM f g = (|fmapAlgebra|)
  where fmapAlgebra (Var a)  = construct (Var (g a))
        fmapAlgebra (Term x) = construct (Term x)

return_FreeM f :: a → FreeM f a
return_FreeM f a = construct (Var a)

join_FreeM f :: FreeM f (FreeM f a) → FreeM f b
join_FreeM f = (|joinAlgebra|)
  where joinAlgebra (Var x)  = x
        joinAlgebra (Term x) = construct (Term x)

wrap_f :: f (FreeM f a) → FreeM f a
wrap_f x = construct (Term x)

⟪−⟫ :: (f a → m a) → FreeM f a → m a
⟪g⟫ = (|extAlgebra|)
  where extAlgebra (Var a)  = return_m a
        extAlgebra (Term x) = join_m (g x)
```

Fig. 1. Constructing free monads via f-algebras

the interpretation of concrete effects. As we have observed in the case of list append in Section 5, the mingling of such concerns can lead to unnecessarily complicated reasoning. Fortunately, a conceptually simpler solution is available: we take the coproduct of the free monad for the functor f that describes our abstract effects with the monad m that describes our concrete effects. We define the coproduct of two monads in Section 9.3 below, and demonstrate how monad coproducts cleanly combine abstract effects with concrete effects. Before that, in the next section, we demonstrate how to construct free monads from initial f-algebras.

9.2 Constructing free monads, via f-algebras

Figure 1 demonstrates how the free monad interface we defined in the previous section may be implemented in terms of initial $(FreeMF\ f\ a)$-algebras, where the functor $FreeMF\ f\ a$ is also defined in Figure 1. The key idea is that a value of type $FreeM\ f\ a$ is constructed

from layers of "terms" described by the functor f, represented by Term constructor, and terminated by "variables", represented by the Var constructor.

The definition of the free monad structure is relatively straightforward, using the functions induced by the initial algebra property of $\mu(\mathit{FreeMF}\ f\ a)$. Each of the properties required of free monads is proved by making use of Proof Principle 1. When we construct the coproduct of a free monad with an arbitrary monad in Section 9.4 we will be able to reuse many of the definitions in Figure 1.

9.3 Coproducts of monads

In the introduction to this section, we intuitively described monad coproducts as the "least commitment" combination of a pair of monads. Formally, this least commitment aspect is realised as the existence of a *unique* monad morphism out of a coproduct for every way of interpreting its consistuent parts. Coproducts of monads are precisely coproducts in the category of monads and monad morphisms. The following definition sets out the precise conditions:

Definition 13
Let $(m_1, \mathit{fmap}_{m_1}, \mathit{return}_{m_1}, \mathit{join}_{m_1})$ and $(m_2, \mathit{fmap}_{m_1}, \mathit{return}_{m_2}, \mathit{join}_{m_2})$ be a pair of monads. A *coproduct* of these two monads is a monad $(m_1{+}m_2, \mathit{fmap}_{m_1+m_2}, \mathit{return}_{m_1+m_2}, \mathit{join}_{m_1+m_2})$ along with a pair of monad morphisms:

$$\begin{array}{lcl} \mathit{inj}_1 & :: & m_1\ a \to (m_1{+}m_2)\ a \\ \mathit{inj}_2 & :: & m_1\ a \to (m_1{+}m_2)\ a \end{array}$$

and the property that for any monad $(m, \mathit{fmap}_m, \mathit{return}_m, \mathit{join}_m)$ and pair of monad morphisms $g_1 : m_1\ a \to m\ a$ and $g_2 : m_2\ a \to m\ a$ there is a *unique* monad morphism $[g_1, g_2] : (m_1{+}m_2)\ a \to m\ a$ such that

$$\begin{array}{lcl} [g_1, g_2] \circ \mathit{inj}_1 & = & g_1 \\ [g_1, g_2] \circ \mathit{inj}_2 & = & g_2 \end{array}$$

In Section 9.1, we demonstrated how the free monad over a functor describing read commands allowed us to provide multiple interpretations of "reading". The monad coproduct $(\mathit{FreeM}\ (\mathit{ReaderF}\ \mathit{String}))+\mathit{IO}$ freely combines the abstract read commands described by the functor $\mathit{ReaderF}\ \mathit{String}$ with the concrete input/output actions of the IO monad. We view $(\mathit{FreeM}\ (\mathit{ReaderF}\ \mathit{String}))+\mathit{IO}$ as the modular reconstruction of the Iteratee monad $\mathit{Reader}\ m\ a$ we presented in Section 1.1.

The following example extends the *swapRead* example from Section 9.1 to perform an input/output effect as well as two abstract read effects. The inj_1 and inj_2 components of the coproduct monad interface allow us to lift effectful computations from the free monad and the IO monad respectively:

$$\begin{array}{l} \mathit{swapRead2} :: ((\mathit{FreeM}\ (\mathit{ReaderF}\ \mathit{String}))+\mathit{IO})\ () \\ \mathit{swapRead2} = \mathbf{do}\ s_1 \leftarrow \mathit{inj}_1\ \mathit{read} \\ \qquad\qquad\quad\ s_2 \leftarrow \mathit{inj}_1\ \mathit{read} \\ \qquad\qquad\quad\ \mathit{inj}_2\ (\mathit{putStrLn}\ (\texttt{"("} +\!\!+ s_2 +\!\!+ \texttt{","} +\!\!+ s_1 +\!\!+ \texttt{")"})) \end{array}$$

This program executes two abstract read commands to read a pair of strings, and then executes a concrete IO action to print the two strings in reverse order to the terminal.

```
type ((FreeM f)+m) a = μ(FreeMF f a|m)

fmap_((FreeM f)+m) :: (a → b) → ((FreeM f)+m) a → ((FreeM f)+m) b
fmap_((FreeM f)+m) g = (|fmapAlgebra|construct_m|)
  where fmapAlgebra (Var a)  = construct_(FreeMF f b) (Var (g a))
        fmapAlgebra (Term x) = construct_(FreeMF f b) (Term x)

return_((FreeM f)+m) :: a → ((FreeM f)+m) a
return_((FreeM f)+m) a = construct_(FreeMF f a) (Var a)

join_((FreeM f)+m) :: ((FreeM f)+m) (((FreeM f)+m) a) → ((FreeM f)+m) b
join_((FreeM f)+m) = (|joinAlgebra|construct_m|)
  where joinAlgebra (Var x)  = x
        joinAlgebra (Term x) = construct (Term x)

inj_1 :: FreeM f a → ((FreeM f)+m) a
inj_1 = ⟪construct_(FreeMF f a) ∘ Term⟫

inj_2 :: m a → ((FreeM f)+m) a
inj_2 = construct_m ∘ fmap_m return_((FreeM f)+m)

[−,−] :: (∀a. FreeM f a → m' a) → (∀a. m a → m' a) → ((FreeM f)+m) a → m' a
[g_1,g_2] = (|caseAlgebra|join_m' ∘ g_2|)
  where caseAlgebra (Var a)  = return_m' a
        caseAlgebra (Term x) = join_m' (⟪g_1⟫^-1 x)
```

Fig. 2. Construction of coproducts with free monads via f-and-m-algebras

We can provide an interpretation for the abstract *read* operations by combining the coproduct interface with the free monad interface. For example, to interpret the read commands as reading from the terminal, we use the *useGetLine* interpretation from Section 9.1:

$$[\langle\!\langle \mathit{useGetLine} \rangle\!\rangle, \mathit{id}] :: ((\mathit{FreeM}\ (\mathit{ReaderF}\ \mathit{String}))+\mathit{IO})\ a \to \mathit{IO}\ a$$

Alternatively, we can interpret the abstract *read* commands as reading from a file handle. The function *useFileHandle* describes how to execute single reads on a file handle as an *IO* action:

```
useFileHandle :: Handle → ReaderF String a → IO a
useFileHandle h (Read k) = do {s ← hGetLine h; return (k s)}
```

Again, we can combine the free monad and monad coproduct interfaces to extend this interpretation of individual abstract *read* commands to all sequences of *read* commands interleaved with arbitrary *IO* actions:

$$\lambda h.\ [\langle\!\langle \mathit{useFileHandle}\ h \rangle\!\rangle, \mathit{id}] :: \mathit{Handle} \to ((\mathit{FreeM}\ (\mathit{ReaderF}\ \mathit{String}))+\mathit{IO})\ a \to \mathit{IO}\ a$$

9.4 Constructing coproducts with free monads via f-and-m-algebras

Figure 2 demonstrates the construction of the coproduct of a free monad with an arbitrary monad m in terms of initial f-and-m-algebras. We program against the abstract interface

of initial f-and-m-algebras, rather relying on any particular implementation. If we take the implementation that we presented in Section 8.2, then the definition of $((\mathit{FreeM}\ f){+}m)\ a$ unfolds to be $m\ (\mathit{Mu}\ (\mathit{FreeMF}\ f\ a : \circ : m))$, which is exactly the Haskell translation of the construction that Hyland *et al.* present.

The definitions of the basic monad structure – *fmap*, *return* and *join* – are almost identical to the corresponding definitions for the free monad in Figure 1. This demonstrates the same feature of the use of f-and-m-algebras that we saw when defining the effectful list append in Section 7: the clean separation of pure and effectful concerns allows us to reuse much of the work we performed in the non-effectful case. The proofs that these definitions actually form a monad carry over just as they did for the list append example.

For the monad coproduct structure, we use the pure and effectful parts of the initial $(\mathit{FreeMF}\ f\ a)$-and-m-algebra stucture – $\mathit{construct}_{\mathit{FreeMF}\ f\ a}$ and $\mathit{construct}_m$ – for the first and second injections inj_1 and inj_2 respectively. Since $\mathit{construct}_{\mathit{FreeMF}\ f\ a}$ injects an single abstract command from f into the coproduct, we use the free monad structure to inject all the commands into the coproduct.

In order for the use of the $(\mathit{FreeMF}\ f\ a)$-and-m-algebra initiality to construct a function on $\mu(\mathit{FreeMF}\ f\ a|m)$ in the definition of $[-,-]$ to be valid, we must check that the second component of $(\!|\mathit{caseAlgebra}|\mathit{join}_{m'} \circ g_2|\!)$ is actually an m-Eilenberg-Moore-algebra. For the first law (Equation 18), we reason as follows:

$$
\begin{array}{ll}
 & \mathit{join}_{m'} \circ g_2 \circ \mathit{return}_m \\
= & \qquad \{g_2 \text{ is a monad morphism (Equation 14)}\} \\
 & \mathit{join}_{m'} \circ \mathit{return}_{m'} \\
= & \qquad \{\text{monad law: } \mathit{join}_{m'} \circ \mathit{return}_{m'} = \mathit{id} \text{ (Equation 8)}\} \\
 & \mathit{id}
\end{array}
$$

The second law (Equation 19) is also straightforward:

$$
\begin{array}{ll}
 & \mathit{join}_{m'} \circ g_2 \circ \mathit{join}_m \\
= & \qquad \{g_2 \text{ is a monad morphism (Equation 15)}\} \\
 & \mathit{join}_{m'} \circ \mathit{join}_{m'} \circ g_2 \circ \mathit{fmap}_m\ g_2 \\
= & \qquad \{\text{monad law: } \mathit{join}_{m'} \circ \mathit{join}_{m'} = \mathit{join}_{m'} \circ \mathit{fmap}'_m\ \mathit{join}_{m'} \text{ (Equation 10)}\} \\
 & \mathit{join}_{m'} \circ \mathit{fmap}_{m'}\ \mathit{join}_{m'} \circ g_2 \circ \mathit{fmap}_m\ g_2 \\
= & \qquad \{\text{naturality of } g_2\} \\
 & \mathit{join}_{m'} \circ g_2 \circ \mathit{fmap}_m\ \mathit{join}_{m'} \circ \mathit{fmap}_m\ g_2 \\
= & \qquad \{\mathit{fmap}_m \text{ preserves function composition (Equation 2)}\} \\
 & \mathit{join}_{m'} \circ g_2 \circ \mathit{fmap}_m\ (\mathit{join}_{m'} \circ g_2)
\end{array}
$$

The proof that $[g_1, g_2]$ satisfies the conditions specified in Definition 13 is remarkably similar to the proof that $(\!|g|\!)$ satisfies the required properties for the free monad specification. This is another testament to the power of f-and-m-algebras.

References

Adámek, J., Bowler, N., Levy, P.B., & Milius, S. (2012). Coproducts of monads on set. *Pages 45–54 of:* Dershowitz, N. (ed), *Proceedings of the 27th Annual ACM/IEEE Symposium on Logic In Computer Science, LICS 2012*.

Atkey, R., Ghani, N., Jacobs, B., & Johann, P. (2012). Fibrational induction meets effects. *Pages 42–57 of:* Birkedal, Lars (ed), *Foundations of Software Science and Computational Structures, FoSSaCS 2012*. Lecture Notes in Computer Science, vol. 7213.

Benton, N., Hughes, J., & Moggi, E. (2000). Monads and effects. *Pages 42–122 of:* Barthe, G., Dybjer, P., Pinto, L., & Saraiva, J. (eds), *Applied Semantics, International Summer School, APPSEM 2000*. Lecture Notes in Computer Science, vol. 2395.

Bird, R. S., & de Moor, O.(1997). *Algebra of programming*. Prentice Hall.

Filinski, A. (1999). Representing layered monads. *Pages 175–188 of:* Appel, A. W., & Aiken, A. (eds), *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999*.

Filinski, A., & Støvring, K. (2007). Inductive reasoning about effectful data types. *Pages 97–110 of:* Hinze, R., & Ramsey, N. (eds), *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*.

Fokkinga, M. M. (1994). *Monadic maps and folds for arbitrary datatypes*. Tech. rept. Memoranda Informatica 94-28. University of Twente.

Ghani, N., & Uustalu, T. (2004). Coproducts of ideal monads. *Theoretical Informatics and Applications*, **38**(4), 321–342.

Gibbons, J. (2003). Origami programming. Gibbons, J., & de Moor, O. (eds), *The fun of programming*. Cornerstones in Computing. Palgrave.

Goguen, J.A., Thatcher, J., & Wagner, E. (1978). An initial algebra approach to the specification, correctness and implementation of abstract data types. *Pages 80–149 of:* Yeh, R. (ed), *Current Trends in Programming Methodology*.

Hyland, M., Plotkin, G. D., & Power, J. (2006). Combining effects: sum and tensor. *Theoretical Computer Science*, **357**(1-3), 70–99.

Jacobs, B., & Rutten, J. (2011). A tutorial on (co)algebras and (co)induction. *Pages 38–99 of:* Sangiorgi, D., & Rutten, J. (eds), *Advanced topics in bisimulation and coinduction*. Tracts in Theoretical Computer Science, no. 52. Cambridge University Press.

Kiselyov, O. (2012). Iteratees. *Pages 166–181 of:* Schrijvers, T., & Thiemann, P. (eds), *Functional and Logic Programming - 11th International Symposium, FLOPS 2012*. Lecture Notes in Computer Science, vol. 7294.

Lambek, J. (1968). A fixed point theorem for complete categories. *Mathematische Zeitschrift*, **103**, 151–161.

Lüth, C., & Ghani, N. (2002). Composing monads using coproducts. *Pages 133–144 of:* Wand, Mitchell, & Jones, Simon L. Peyton (eds), *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming, ICFP 2002*.

Mac Lane, S. (1998). *Categories for the working mathematician*. 2nd edn. Graduate Texts in Mathematics, no. 5. Springer-Verlag.

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

Pardo, A. (2004). Combining datatypes and effects. *Pages 171–209 of:* Vene, V., & Uustalu, T. (eds), *Advanced Functional Programming, 5th International School, AFP 2004*. Lecture Notes in Computer Science, vol. 3622.

Peyton Jones, S. L., & Wadler, P. (1993). Imperative functional programming. *Pages 71–84 of:* Deusen, M. S. Van, & Lang, B. (eds), *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1993*.

Piróg, M., & Gibbons, J. (2012). Tracing monadic computations and representing effects. *Pages 90–111 of:* Chapman, J., & Levy, P. B. (eds), *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP 2012*. Electronic Proceedings in Theoretical Computer Science, vol. 76.

Sheard, T., & Pasalic, E. (2004). Two-level types and parameterized modules. *Journal of Functional Programming*, **14**(5), 547–587.

Swierstra, W. (2008). Data types à la carte. *Journal of Functional Programing*, **18**(4), 423–436.

Swierstra, W., & Altenkirch, T. (2007). Beauty in the beast. *Pages 25–36 of:* Keller, G. (ed), *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007*.