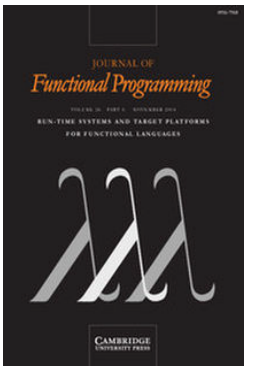

Interleaving data and effects

ROBERT ATKEY and PATRICIA JOHANN

Journal of Functional Programming / Volume 25 / 2015 / e20
DOI: 10.1017/S0956796815000209, Published online: 20 November 2015

JFP **25**, e20, 44 pages, 2015.
doi:10.1017/S0956796815000209

Interleaving data and effects

ROBERT ATKEY
University of Strathclyde, Glasgow, Scotland
(*e-mail:* robert.atkey@strath.ac.uk)

PATRICIA JOHANN
Appalachian State University, Boone, North Carolina, USA
(*e-mail:* johannp@appstate.edu)

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, and lazily constructed infinite values can be represented by pure data interleaved with the possibility of non-terminating computation. Straightforward application of initial algebra techniques to effectful datatypes leads either to unsound conclusions if we ignore the possibility of effects, or to unnecessarily complicated reasoning because the pure and effectful concerns must be considered simultaneously. We show how pure and effectful concerns can be separated using 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.

Initial f-and-m-algebras are originally due to Filinski and Støvring, who studied them in the category Cpo. They were subsequently generalised to arbitrary categories by Atkey, Ghani, Jacobs, and Johann in a FoSSaCS 2012 paper. In this article we aim to introduce the general 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:

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$

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 f-algebras – i.e., least fixed points of functors f – and combinators such as *foldr* are derived from the universal properties of initial f-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 f-algebra methodology does *not* provide the right level of abstraction. Specifically, we consider recursive datatypes that interleave pure data with effectful computation[1]. For example, lists of characters that are interleaved with input/output operations that read them from an external source can be described by the following datatype declaration:

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

Similarly, as we shall see in Section 1.1, Haskell's lazy datatypes can be thought of as pure data interleaved with the possibility of non-termination effects.

Using the initial f-algebra methodology 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 4). 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 describes the pure part of the datatype, and m is a monad that describes the effects. In this article we will show that initial f-and-m-algebras represent a better level of abstraction for dealing with interleaved data and effects.

The key idea behind f-and-m-algebras is to separate the concerns of dealing with pure data, via f-algebras, and effects, via m-Eilenberg–Moore-algebras. For readers unfamiliar with m-Eilenberg–Moore-algebras, an m-Eilenberg–Moore-algebra can be thought of as a special kind of f-algebra that interacts well with computational

[1] Following Filinski & Støvring (2007), we will refer to datatypes that interleave pure data with effects as *effectful datatypes*. Strictly speaking, this is a misnomer because values of these types only contain pure descriptions of effects. However, we feel that this name correctly conveys the intuition that datatypes with interleaved monadic values are of a different character to datatypes without monadic values, as we illustrate by the examples in this introduction.

effects described by a monad m. We will introduce m-Eilenberg–Moore-algebras properly in Section 5.1.

We shall see in Section 6 that the separation into pure and effectful concerns 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. Moreover, the new definitions are generic in the interleaved monad we use for representing effects — for example, the *IO* monad for input/output effects, or the non-termination monad for laziness.
- *Proofs* about functions on interleaved datatypes also carry over almost unchanged from their pure counterparts. We demonstrate this through the proof of associativity for append on effectful lists, again generic in the monad representing the effects. The proof 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 in the specific setting of Cpo (the category of complete partial orders and continuous functions) (Filinski & Støvring, 2007), and was subsequently extended to a general category-theoretic setting for arbitrary functors f by 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 and show how they can be used to structure and reason about functional programs in practice, without the heavy category-theoretic prerequisites of Atkey *et al.*'s work.

1.1 Interleaving data and effects

To motivate our consideration of interleaved data and effects, in this section we give two scenarios where Haskell *implicitly* interleaves effects with pure data. By making this implicit interleaving explicit, we will see in the main body of this article how the f-and-m-algebra formalism allows for the implicit assumptions made when reasoning about Haskell datatypes using initial f-algebras can be made explicit as well.

1.1.1 I/O effects

The *hGetContents* function from the Haskell standard library provides an example of implicit interleaving of data with input/output effects. The *hGetContents* function has the following type:

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

Reading the type of this function, we might assume that it operates by reading all the available data from the file referenced by the given handle as an *IO* action,

yielding the list of characters as pure data. In fact, the standard implementation of this function postpones the reading of data from the handle 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*, with 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.

Using the $List_{io}$ types defined on Page 2, we can give an implementation of *hGetContents* whose type makes explicit the interleaving of data and effects. A simple implementation can be given in terms of the standard Haskell primitives for performing IO on file handles:

```
hGetContents :: Handle → List_io
hGetContents h = List_io (do isEOF ← hIsEOF h
                             if isEOF then return Nil_io
                             else do c ← hGetChar h
                                     return (Cons_io c (hGetContents h)))
```

By using the $List_{io}$ 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.

We return to the example of interleaved I/O effects in Section 8, where we will see how practical techniques that have been proposed by the Haskell community for making the interleaving explicit can be handled neatly by using f-and-m-algebras.

1.1.2 Non-termination

A second scenario involving implicitly interleaved effects is built in to every Haskell type: the possibility of non-termination while inspecting a pure value. Haskell has a

non-strict semantics, which is usually implemented using a lazy evaluation strategy, in which the computation of a value is only invoked if the value is actually needed. For the purposes of reasoning about the behavior of Haskell programs, we can model the possibility of non-termination using the *lifting* or *non-termination* monad, $(-)_\perp$. This monad adds a bottom element $\perp$ to a type, representing the possibility of non-termination at that type. Every Haskell type is implicitly lifted using this monad.

A well-known benefit of Haskell's implicit possibility of non-termination at every type is the easy representation of infinite data structures. Laziness means that a computation that generates an infinite value is evaluated on demand as the structure is explored. We implicitly used this facility in our definition of *hGetContents* above to deal with the possibility of *Handle*s that may return infinite streams of values. Unfortunately, the beneficial capability of representing infinite data structures comes with the downside that we can no longer distinguish, just by looking at the types, between finite lists and possibly infinite lists. Both are assigned the type $[a]$ for some a. It is often the case that functions are written under the implicit assumption that they are only applied to finite lists (for example, the standard *reverse* function). Likewise, when reasoning about Haskell programs it is often implicitly assumed that lists are finite, so that standard techniques like induction can be applied. We examine the assumptions implicit in the *reverse* function in Section 6.2.

To make these implicit assumptions explicit, we can modify the type $List_{io}$ from the introduction to get the type $List_{lazy}$ of lists interleaved with the possibility of non-termination:

$$\begin{array}{ll} \textbf{data}\ List'_{lazy}\ a & \textbf{newtype}\ List_{lazy}\ a = \\ \quad = \mathsf{Nil}_{lazy} & \quad \mathsf{List}_{lazy}\ (List'_{lazy}\ a)_\perp \\ \quad \mid \mathsf{Cons}_{lazy}\ a\ (List_{lazy}\ a) & \end{array}$$

A value of type $List_{lazy}\ a$ is thus a possibly non-terminating computation that results in either a Nil_{lazy} constructor, or Cons_{lazy} constructor applied to a value of type a and another $List_{lazy}\ a$. (Note that, for simplicity's sake, we have not modelled the fact that constructors of datatypes in Haskell also evaluate their arguments lazily.)

The $List_{lazy}\ a$ type is precisely the 'even' style of interleaving pure data and laziness advocated by Wadler *et al.* (1998). The obvious alternative interleaving, named the "odd" style by Wadler *et al.*, is expressible as a single datatype declaration:

$$\begin{array}{l} \textbf{data}\ List_{odd}\ a \\ \quad = \mathsf{Nil}_{odd} \\ \quad \mid \mathsf{Cons}_{odd}\ a\ (List_{odd}\ a)_\perp \end{array}$$

In the "odd" formulation, the lifting monad $(-)_\perp$ is only used in the recursive position in the Cons_{odd} constructor. Wadler *et al.* argue that this "odd" style leads to lazy computations being forced much earlier than the programmer might expect: since the first element of a list in the odd style is *not* wrapped in the lifting monad, any function that returns a $List_{odd}\ a$ value must always have the first element available immediately. The 'even' style, as exemplified by our $List_{lazy}$ type constructor above, is, Wadler *et al.* argue, usually what is expected.

Correctly reasoning about values of type $List_{lazy}\ a$ and other lazy data structures has traditionally required the use of domain-theoretic techniques (Pitts (1996) provides a comprehensive overview). The technique of using f-and-m-algebras that we present in this article allows sound reasoning about lazy data structures at an abstract level, dispensing with the need to directly invoke domain-theoretic concepts. Indeed, Filinski and Støvring used lazy data structures as their initial motivation for introducing f-and-m-algebras in the category Cpo (Filinski & Støvring, 2007).

1.1.3 A common generalisation

We have now seen two scenarios in which list-like datatypes with interleaved effects naturally arise, namely the $List_{io}$ datatype from Page 2 and the $List_{lazy}$ datatype above. The obvious common generalisation abstracts over the monad m:

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

A value of type $List\ m\ a$ consists of an effect described by m, then either a Nil_m to indicate the end of the list, or a 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 now generalise further by replacing the constructors Nil_m and Cons_m with an arbitrary functor f that describes the data to be interleaved with the effects of the monad m. Doing so, we arrive at the following definition:

$$\begin{array}{ll} \textbf{data}\ MuFM'_0\ f\ m & \textbf{newtype}\ MuFM_0\ f\ m = \\ \quad = \mathsf{In}\ (f\ (MuFM_0\ f\ m)) & \quad \mathsf{Mu}\ (m\ (MuFM'_0\ f\ m)) \end{array}$$

(We have named these types $MuFM_0\ f\ m$ and $MuFM'_0\ f\ m$ with a 0 subscript because we will introduce a more refined, but isomorphic, presentation in Section 7.2.) This definition makes it clear that the datatypes we are considering interleave pure data, represented by the functor f, with effects, represented by the monad m. Our definition is the generalisation of Wadler *et al.*'s 'even'-style lazy lists, from lists to arbitrary functors f, and from lifting to arbitrary monads m.

The aim of this article is to show that f-and-m-algebras are the appropriate level of abstraction both for defining functions that operate on values of type $MuFM_0\ f\ m$, and for reasoning about them.

1.2 The contents 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:

- In Section 2, we recall the standard definitions of f-algebras, initial f-algebras, and monads, all in a functional programming context. We highlight the proof principle associated with initial f-algebras (Proof Principle 1), and demonstrate

that f-algebras can be thought of as abstract interfaces 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.
- To motivate the use of f-and-m-algebras, in Section 4 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 5, 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 6, 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 7, 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 arbitrary functors f and monads m.
- In Section 8, we present an extended example of the use of initial f-and-m-algebras. Motivated by the undesirable properties of implicitly interleaving pure lists with I/O effects that we described in the previous section, the Haskell community has developed several approaches that explicitly interleave effects with data. Examples include Kiselyov's Iteratees (Kiselyov, 2012) and Gonzalez's `pipes` library[2]. We show that at least these two constructions are instances of the general construction of the coproduct of a free monad with another monad. Hyland, Plotkin and Power previously gave this coproduct construction using purely categorical techniques. In Section 8, we reconstruct this result using f-and-m-algebras. Several of the properties proved by Kiselyov and Gonzalez for their respective libraries are shown to follow directly from the observation that their definitions are instances of the sum of a free monad with another monad.

2 Background: f-algebras, initial f-algebras, and monads

Initial f-and-m-algebras build upon the foundations of initial f-algebras, and of monads. We recall the definition of f-algebras, initial f-algebras, and monads in this section, and derive the accompanying definitional and proof principles. We will

[2] http://hackage.haskell.org/package/pipes

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:

```
class Functor f where
  fmap :: (a → b) → f a → 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 an operation for reducing an f-structure of values to a value. Formally, f-algebras are defined as follows:

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

Given a pair of f-algebras, there is also the concept of a homomorphism (*i.e.*, a structure preserving map) between them:

Definition 3
Given a pair of f-algebras (a, k_a) and (b, k_b), an *f-algebra homomorphism* between them is a function $h :: a \to b$ such that the following diagram commutes[3]:

$$\begin{array}{ccc} f\ a & \xrightarrow{fmap_f\ h} & f\ b \\ {\scriptstyle k_a}\downarrow & & \downarrow{\scriptstyle k_b} \\ a & \xrightarrow{h} & b \end{array} \tag{3}$$

[3] By *commutes*, we mean the standard meaning: the two paths in the diagram denote equal functions built by composing the labels on the arrows.

Definition 4
An *initial f-algebra* is an f-algebra $(\mu f, in)$ such that for any f-algebra (a,k), there exists a unique f-algebra homomorphism from $(\mu f, in)$ to (a,k). We write this homomorphism as $(\!|k|\!)$, and note that $(\!|k|\!)$ is a function of type $\mu f \to a$ such that $(\!|k|\!) \circ in = k \circ fmap_f\ (\!|k|\!)$.

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 $(\mu f, in)$ is an initial f-algebra.

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

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

holds if and only if g is an f-algebra homomorphism:

$$g \circ in = k \circ fmap_f\ g.$$

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 & 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:

```
data ListF a x          fmap_(ListF a) :: (x → y) → ListF a x → ListF a y
  = Nil                 fmap_(ListF a) g Nil          = Nil
  | Cons a x            fmap_(ListF a) g (Cons a x) = Cons a (g x)
```

Assuming for the moment that the Haskell datatype $[a]$ only contains *finite* lists, the following definitions witness that $[a]$ is the carrier of an initial *ListF a* algebra:

```
in :: ListF a [a] → [a]
in Nil          = [ ]
in (Cons a xs) = a : xs
```

and

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

As we pointed out in Section 1.1, the assumption that the type $[a]$ only contains finite lists is unsound. We have failed to account for the possibility of non-termination effects interleaved between the elements of the list. With extra effort, it is possible to integrate non-termination effects into the f-algebra formalism, as we show in Section 4. However, in Section 5 we show how f-and-m-algebras offer a simple and direct solution to reasoning about Haskell's lazy lists, as well as other datatypes interleaved with effects.

Example 2
Again ignoring the possibility of non-termination, we can implement the carrier of an initial f-algebra for an arbitrary functor (f, fmap_f) as a recursive datatype:

$$\mathbf{data}\ \mathit{Mu}\ f = \mathsf{In}\ \{\mathit{unIn} :: f\ (\mathit{Mu}\ f)\} \tag{4}$$

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:

$$\begin{aligned}&\mathit{in} :: f\ (\mathit{Mu}\ f) \to \mathit{Mu}\ f\\&\mathit{in} = \mathsf{In}\end{aligned}$$

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:

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

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 $\mathit{Mu}\ (\mathit{ListF}\ a)$ are not identical, but they are both initial $(\mathit{ListF}\ a)$-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 $\mathit{Mu}\ f$. 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 them, and Proof Principle 1 to prove that the translations are mutually inverse. This isomorphism result is known as Lambek's Lemma (Lambek, 1968).

2.3 Monads

As is standard in Haskell programming, we describe effectful computations in terms of monads (Moggi, 1991; Peyton Jones & Wadler, 1993). 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 because the categorical definition is more convenient for equational reasoning. Standard references such as the lecture notes by 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:

$$join_m \circ return_m = id \tag{5}$$

$$join_m \circ fmap_m\ return_m = id \tag{6}$$

$$join_m \circ fmap_m\ join_m = join_m \circ join_m \tag{7}$$

and also the naturality laws:

$$return_m \circ f = fmap_m\ f \circ return_m \tag{8}$$

$$join_m \circ fmap_m\ (fmap_m\ f) = fmap_m\ f \circ join_m \tag{9}$$

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 two 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 8.

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 two monads. A *monad morphism* between them is a function $h :: m_1\ a \to m_2\ a$ such that:

$$h \circ fmap_{m_1}\ g = fmap_{m_2}\ g \circ h \tag{10}$$

$$h \circ return_{m_1} = return_{m_2} \tag{11}$$

$$h \circ join_{m_1} = join_{m_2} \circ h \circ fmap_{m_1}\ h \tag{12}$$

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. (Purity here means that this proof does *not* apply to Haskell's lazy lists, unlike the proofs we will present in

Sections 4 and 6.) In Section 4 we attempt to use the initial f-algebra technique to prove the analogous property in a setting with interleaved effects, and see that direct use of initial f-algebras makes the definition and proof unnecessarily complicated. In Section 5, 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 f-algebras. Hence we assume that an initial $(ListF\ a)$-algebra $(\mu(ListF\ a), in)$ exists, and we write $(\!|-|\!)$ for the unique homomorphism induced by initiality, *i.e.*, for the unique map taking each f-algebra (a, k) to the unique f-algebra homomorphism $(\!|k|\!) :: \mu f \to a$. We can define *append* in terms of $(\!|-|\!)$ as:

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

Immediately from the definition of *append* we know that $\lambda xs.\ append\ xs\ ys$ is a $(ListF\ a)$-algebra homomorphism, for any ys, 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 $in\ \mathsf{Nil}$ and $in\ (\mathsf{Cons}\ a\ xs)$. We have:

$$append\ (in\ \mathsf{Nil})\ ys = ys \tag{13}$$

$$append\ (in\ (\mathsf{Cons}\ a\ xs))\ ys = in\ (\mathsf{Cons}\ a\ (append\ xs\ ys)) \tag{14}$$

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:

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

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

$$k\ \mathsf{Nil} = append\ ys\ zs \tag{15}$$

$$k\ (\mathsf{Cons}\ a\ xs) = in\ (\mathsf{Cons}\ a\ xs) \tag{16}$$

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

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

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:

$$\begin{array}{ll} & append\ (append\ (in\ \mathsf{Nil})\ ys)\ zs \\ = & \qquad \{\text{Equation (13)}\} \\ & append\ ys\ zs \\ = & \qquad \{\text{definition of } k \text{ (Equation (15))}\} \\ & k\ \mathsf{Nil} \\ = & \qquad \{\text{definition of } fmap_{ListF\ a}\} \\ & k\ (fmap_{ListF\ a}\ (\lambda xs.\ append\ (append\ xs\ ys)\ zs)\ \mathsf{Nil}) \end{array}$$

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

$$\begin{array}{ll} & append\ (append\ (in\ (\mathsf{Cons}\ a\ xs))\ ys)\ zs \\ = & \qquad \{\text{Equation (14)}\} \\ & append\ (in\ (\mathsf{Cons}\ a\ (append\ xs\ ys)))\ zs \\ = & \qquad \{\text{Equation (14)}\} \\ & in\ (\mathsf{Cons}\ a\ (append\ (append\ xs\ ys)\ zs)) \\ = & \qquad \{\text{definition of } k \text{ (Equation (16))}\} \\ & k\ (\mathsf{Cons}\ a\ (append\ (append\ xs\ ys)\ zs)) \\ = & \qquad \{\text{definition of } fmap_{ListF\ a}\} \\ & k\ (fmap_{ListF\ a}\ (\lambda xs.\ append\ (append\ xs\ ys)\ zs)\ (\mathsf{Cons}\ a\ xs)) \end{array}$$

□

Thus the proof that *append* is associative is relatively straightforward, using Proof Principle 1. We shall see below, in Section 4, 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 5 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 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*. 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 these complications through an extension of the list append example from Section 3 to the case of lists with interleaved effects. We carry out this proof directly at the level of f-algebras, just as we did in the previous section. After the proof, we reflect on the difficulties that we encountered in the proof. Some of these difficulties can be mitigated by use of more advanced f-algebra techniques,

such as *fold fusion*. However, we will discover that f-and-m-algebras yield a more satisfactory solution.

Our presentation is parametric 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 declaration of $List'\ m\ a$, and comparing it 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 $(\mu(ListF\ a \circ m), in)$ exists, and we write $(\!|-|\!)$ for the unique homomorphism induced by initiality. We now define *eAppend* ("*e*" for effectful) by:

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

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 manage the 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 oblivious to the presence of effects.

As we did for *append* in Equations (13) and (14) above, we can derive two properties of *eAppend*. Equations (17) and (18) tell us how *eAppend* acts on pure computations that return values constructed with each of the list constructors:

$$
eAppend\ (return_m\ (in\ \mathsf{Nil}))\ ys = ys \tag{17}
$$

and

$$
\begin{aligned}
&eAppend\ (return_m\ (in\ (\mathsf{Cons}\ a\ xs)))\ ys \\
=\ &return_m\ (in\ (\mathsf{Cons}\ a\ (eAppend\ xs\ ys)))
\end{aligned} \tag{18}
$$

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 (17) 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. We use k_l to denote an instance of the function k defined in the body of *eAppend* with the free variable ys replaced by l.

Let us simplify the left hand side of the equation to be proved:

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

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

$$\begin{array}{ll} & eAppend\ (eAppend\ xs\ ys)\ zs \\ = & \qquad \{\text{definition of } eAppend\} \\ & eAppend\ (join_m\ (fmap_m\ (\!|k_{ys}|\!)\ xs))\ zs \\ = & \qquad \{\text{definition of } eAppend\} \\ & join_m\ (fmap_m\ (\!|k_{zs}|\!)\ (join_m\ (fmap_m\ (\!|k_{ys}|\!)\ xs))) \\ = & \qquad \{\text{naturality of } join_m \text{ (Equation (9))}\} \\ & join_m\ (join_m\ (fmap_m\ (fmap_m\ (\!|k_{zs}|\!))\ (fmap_m\ (\!|k_{ys}|\!)\ xs))) \\ = & \qquad \{\text{monad law: } join_m \circ join_m = join_m \circ fmap_m\ join_m \text{ Equation (7)}\} \\ & join_m\ (fmap_m\ join_m\ (fmap_m\ (fmap_m\ (\!|k_{zs}|\!))\ (fmap_m\ (\!|k_{ys}|\!)\ xs))) \\ = & \qquad \{fmap_m \text{ preserves composition (Equation (2))}\} \\ & join_m\ (fmap_m\ (join_m \circ fmap_m\ (\!|k_{zs}|\!) \circ (\!|k_{ys}|\!))\ xs) \\ = & \qquad \{\text{definition of } eAppend\} \\ & join_m\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ (\!|k_{ys}|\!))\ xs) \end{array}$$

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

$$(\!|k_{eAppend\ ys\ zs}|\!) = (\lambda l.\ eAppend\ l\ zs) \circ (\!|k_{ys}|\!) \tag{19}$$

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

$$\begin{array}{ll} & eAppend\ ((\!|k_{ys}|\!)\ (in\ x))\ zs \\ = & k_{eAppend\ xs\ ys}\ (fmap_{ListF\ a}\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ (\!|k_{ys}|\!)))\ x) \end{array}$$

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. Note that, we are unable to directly apply our knowledge of the effect of *eAppend* on Nil (Equation (17)), unlike in the proof of Theorem 1 where we could use Equation (13). This is because we had to unfold

the definition of *eAppend* in order to apply Proof Principle 1.

$$
\begin{array}{ll}
 & eAppend\ (\llparenthesis k_{ys} \rrparenthesis\ (in\ \mathsf{Nil}))\ zs \\
= & \qquad \{\llparenthesis k_{ys} \rrparenthesis \text{ is a } (ListF\ a \circ m)\text{-algebra homomorphism}\} \\
 & eAppend\ (k_{ys}\ (fmap_{ListF\ a}\ (fmap_m\ \llparenthesis k_{ys} \rrparenthesis)\ \mathsf{Nil}))\ zs \\
= & \qquad \{\text{definition of } fmap_{ListF\ a}\} \\
 & eAppend\ (k_{ys}\ \mathsf{Nil})\ zs \\
= & \qquad \{\text{definition of } k_{ys}\} \\
 & eAppend\ ys\ zs \\
= & \qquad \{\text{definition of } k_{eAppend\ ys\ zs}\} \\
 & k_{eAppend\ ys\ zs}\ \mathsf{Nil} \\
= & \qquad \{\text{definition of } fmap_{ListF\ a}\} \\
 & k_{eAppend\ xs\ ys}\ (fmap_{ListF\ a}\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ \llparenthesis k_{ys} \rrparenthesis))\ \mathsf{Nil})
\end{array}
$$

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

$$
\begin{array}{ll}
 & eAppend\ (\llparenthesis k_{ys} \rrparenthesis\ (in\ (\mathsf{Cons}\ a\ xs)))\ zs \\
= & \qquad \{\llparenthesis k_{ys} \rrparenthesis \text{ is a } (ListF\ a \circ m)\text{-algebra homomorphism}\} \\
 & eAppend\ (k_{ys}\ (fmap_{ListF\ a}\ (fmap_m\ \llparenthesis k_{ys} \rrparenthesis)\ (\mathsf{Cons}\ a\ xs)))\ zs \\
= & \qquad \{\text{definition of } fmap_{ListF\ a}\} \\
 & eAppend\ (k_{ys}\ (\mathsf{Cons}\ a\ (fmap_m\ \llparenthesis k_{ys} \rrparenthesis\ xs)))\ zs \\
= & \qquad \{\text{definition of } k_{ys}\} \\
 & eAppend\ (return_m\ (in\ (\mathsf{Cons}\ a\ (join_m\ (fmap_m\ \llparenthesis k_{ys} \rrparenthesis\ xs)))))\ zs \\
= & \qquad \{\text{definition of } eAppend\} \\
 & eAppend\ (return_m\ (in\ (\mathsf{Cons}\ a\ (eAppend\ xs\ ys))))\ zs \\
= & \qquad \{\text{Equation (18)}\} \\
 & return_m\ (in\ (\mathsf{Cons}\ a\ (eAppend\ (eAppend\ xs\ ys)\ zs))) \\
= & \qquad \{\text{definition of } eAppend\} \\
 & return_m\ (in\ (\mathsf{Cons}\ a \\
 & \qquad (join_m\ (fmap_m\ \llparenthesis k_{zs} \rrparenthesis\ (join_m\ (fmap_m\ \llparenthesis k_{ys} \rrparenthesis\ xs)))))) \\
= & \qquad \{\text{naturality of } join_m \text{ (Equation (9))}\} \\
 & return_m\ (in\ (\mathsf{Cons}\ a \\
 & \qquad (join_m\ (join_m\ (fmap_m\ (fmap_m\ \llparenthesis k_{zs} \rrparenthesis)\ (fmap_m\ \llparenthesis k_{ys} \rrparenthesis\ xs)))))) \\
= & \qquad \{\text{monad law: } join_m \circ join_m = join_m \circ fmap_m\ join_m \text{ (Equation (7))}\} \\
 & return_m\ (in\ (\mathsf{Cons}\ a \\
 & \qquad (join_m\ (fmap_m\ join_m \\
 & \qquad\quad (fmap_m\ (fmap_m\ \llparenthesis k_{zs} \rrparenthesis)\ (fmap_m\ \llparenthesis k_{ys} \rrparenthesis\ xs)))))) \\
= & \qquad \{fmap_m \text{ preserves function composition (Equation (2))}\} \\
 & return_m\ (in\ (\mathsf{Cons}\ a \\
 & \qquad (join_m\ (fmap_m\ (join_m \circ fmap_m\ \llparenthesis k_{zs} \rrparenthesis \circ \llparenthesis k_{ys} \rrparenthesis)\ xs)))) \\
= & \qquad \{\text{definition of } eAppend\} \\
 & return_m\ (in\ (\mathsf{Cons}\ a \\
 & \qquad (join_m\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ \llparenthesis k_{ys} \rrparenthesis)\ xs))))
\end{array}
$$

$$
\begin{aligned}
&= && \{\text{definition of } k_{eAppend\ ys\ zs}\} \\
&\quad k_{eAppend\ ys\ zs}\ (\mathsf{Cons}\ a\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ (\!|k_{ys}|\!))\ xs)) \\
&= && \{\text{definition of } fmap_{ListF\ a}\} \\
&\quad k_{eAppend\ ys\ zs} \\
&\qquad (fmap_{ListF\ a}\ (fmap_m\ ((\lambda l.\ eAppend\ l\ zs) \circ (\!|k_{ys}|\!)))\ (\mathsf{Cons}\ a\ xs))
\end{aligned}
$$

□

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 (17) 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 (18) 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 occurrences of $join_m$ and $fmap_m$.

To some extent, it is possible to mitigate these problems without using f-and-m-algebras.

The first problem can be addressed by noting that $eAppend\ xs\ ys = extend\ (\!|k_{ys}|\!)\ xs$, where $extend :: (a \to m\ b) \to m\ a \to m\ b$ is the argument flipped bind $(\gg\!=)$ operation for the monad m. Using the general fact that $extend\ f\ (extend\ g\ x) = extend\ (extend\ f \circ g)\ x$ allows for a quicker reduction of the theorem statement to Equation (19).

The second problem can be addressed by using the general *fold fusion law* to prove Equation (19). Fold fusion is an important consequence of Proof Principle 1 that can first be derived as an independent lemma. In the current setting, the fold fusion law can be stated as follows:

$$
\left.
\begin{aligned}
&f\ (k_1\ \mathsf{Nil}) = k_2\ \mathsf{Nil} \\
&f\ (k_1\ (\mathsf{Cons}\ x\ xs)) = k_2\ (\mathsf{Cons}\ x\ (fmap_m\ f\ xs))
\end{aligned}
\right\} \Rightarrow f \circ (\!|k_1|\!) = (\!|k_2|\!)
$$

Using fold fusion shortens the sequences of equational reasoning for the Nil and Cons cases by a few lines at the start and the end, but does not free us from having to unfold the definition of *eAppend* and reason using the monad laws. Using fold fusion and the general property of *extend* does save us a little effort, but does not clearly separate the pure and effectful parts of the proof in the way that the f-and-m-algebra proof principle in the next section will allow us to, and still does not allow us to directly reuse the reasoning from the proof in the pure case in Theorem 1.

We see, then, that the definition and proof that we have given in this section – not to mention alternative proofs akin to those discussed above – demonstrate that

direct use of initial f-algebras provides the wrong level of abstraction for dealing with datatypes that interleave data and effects.

5 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 & Støvring (2007), and generalised to arbitrary functors by 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 m-Eilenberg–Moore algebra. m-Eilenberg–Moore algebra structure for a type a describes how to incorporate the effects of the monad m into values of type a.

5.1 m-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, l) of a type a and a function

$$l :: m\ a \to a$$

such that the following two diagrams commute:

$$\begin{array}{ccc} a & \xrightarrow{return_m} & m\ a \\ & \searrow^{id} & \downarrow{\scriptstyle l} \\ & & a \end{array} \tag{20}$$

$$\begin{array}{ccc} m\ (m\ a) & \xrightarrow{join_m} & m\ a \\ {\scriptstyle fmap_m\ l}\downarrow & & \downarrow{\scriptstyle l} \\ m\ a & \xrightarrow{\quad l \quad} & a \end{array} \tag{21}$$

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

In terms of computational effects, an m-Eilenberg–Moore-algebra (a, l) 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* :

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

$$\begin{array}{l}\mathit{fmap}_{\mathit{ErrorM}}\ g\ (\mathsf{Ok}\ a) = \mathsf{Ok}\ (g\ a) \\ \mathit{fmap}_{\mathit{ErrorM}}\ g\ (\mathsf{Error}\ \mathit{msg}) = \mathsf{Error}\ \mathit{msg}\end{array}$$

$$\mathit{return}_{\mathit{ErrorM}}\ a = \mathsf{Ok}\ a$$

$$\begin{array}{l}\mathit{join}_{\mathit{ErrorM}}\ (\mathsf{Ok}\ (\mathsf{Ok}\ a)) = \mathsf{Ok}\ a \\ \mathit{join}_{\mathit{ErrorM}}\ (\mathsf{Ok}\ (\mathsf{Error}\ \mathit{msg})) = \mathsf{Error}\ \mathit{msg} \\ \mathit{join}_{\mathit{ErrorM}}\ (\mathsf{Error}\ \mathit{msg}) = \mathsf{Error}\ \mathit{msg}\end{array}$$

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

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

The function *throw* and the constructor ErrorCall are part of the *Control.Exception* module in the Haskell standard library. The algebra l 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, captures 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 m-Eilenberg–Moore algebras for a given monad m is the class of *free* m-Eilenberg–Moore-algebras. The *free* m-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 m-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*

$$h :: (a, l_a) \to (b, l_b)$$

consists of a function $h :: a \to b$ such that the following diagram commutes:

$$\begin{array}{ccc} m\ a & \xrightarrow{\mathit{fmap}_m\ h} & m\ b \\ {\scriptstyle l_a}\downarrow & & \downarrow{\scriptstyle l_b} \\ a & \xrightarrow{h} & b \end{array} \tag{22}$$

5.2 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 m-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, k, l) of an object a and two functions:

$$\begin{aligned} k &:: f\ a \to a \\ l &:: m\ a \to a \end{aligned}$$

where l 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*

$$h :: (a, k_a, l_a) \to (b, k_b, l_b)$$

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

$$h \circ k_a = k_b \circ fmap_f\ h \tag{23}$$

$$h \circ l_a = l_b \circ fmap_m\ h \tag{24}$$

Given the above definitions, the definition of initial f-and-m-algebra 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 $(\mu(f|m), in_f, in_m)$ such that for any f-and-m-algebra (a, k, l), there exists a unique f-and-m-algebra homomorphism from $(\mu(f|m), in_f, in_m)$ to (a, k, l). We write this homomorphism as $(\!|k|l|\!)$ and note that $(\!|k|l|\!)$ is a function of type $\mu(f|m) \to a$.

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 $(\mu(f|m), in_f, in_m)$ is an initial f-and-m-algebra.

Let (a, k, l) be an f-and-m-algebra, and let $(\!|k|l|\!)$ denote the induced function of type $\mu(f|m) \to a$. For any function $g :: \mu(f|m) \to a$, the equation:

$$(\!|k|l|\!) = g$$

holds if and only if

$$g \circ in_f = k \circ fmap_f\ g \tag{25}$$

and

$$g \circ in_m = l \circ fmap_m\ g \tag{26}$$

The key feature of Proof Principle 2 is that it cleanly splits the pure (Equation (25)) and effectful (Equation (26)) 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 4. We shall see this separation in action for our list append running example in the next section.

Example 3
The *List m a* datatype in the introduction was defined as follows:

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

This datatype can be presented as the carrier of an initial $(ListF\ a)$-and-m-algebra. The $in_{ListF\ a}$ function is defined as follows:

$$\begin{array}{l} in_{ListF\ a} :: ListF\ a\ (List\ m\ a) \to List\ m\ a \\ in_{ListF\ a}\ \mathsf{Nil} \qquad\quad = \mathsf{List}\ (return_m\ \mathsf{Nil}_m) \\ in_{ListF\ a}\ (\mathsf{Cons}\ a\ xs) = \mathsf{List}\ (return_m\ (\mathsf{Cons}_m\ a\ xs)) \end{array}$$

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

$$\begin{array}{l} in_m :: m\ (List\ m\ a) \to List\ m\ a \\ in_m\ ml = \mathsf{List}\ (\textbf{do}\ \{\mathsf{List}\ x \leftarrow ml\,; x\}) \end{array}$$

(If it were not for the List constructor, then in_m would simply be $join_m$.)

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 $List\ m\ a$:

$$
\begin{aligned}
&(\!|-|-|\!) :: (ListF\ a\ b \to b) \to (m\ b \to b) \to List\ m\ a \to b \\
&(\!|k|l|\!) = loop \\
&\quad \textbf{where}\ loop :: List\ m\ a \to b \\
&\qquad\quad loop\ (\mathsf{List}\ x) = l\ (fmap_m\ loop'\ x) \\
&\\
&\qquad\quad loop' :: List'\ m\ a \to b \\
&\qquad\quad loop'\ \mathsf{Nil}_m = k\ \mathsf{Nil} \\
&\qquad\quad loop'\ (\mathsf{Cons}_m\ a\ xs) = k\ (\mathsf{Cons}\ a\ (loop\ xs))
\end{aligned}
$$

We will give a general construction of initial f-and-m-algebras in Section 7.2 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.

6 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 4. We use the abstraction of (initial) f-and-m-algebras, firstly to simplify the implementation of *eAppend* from Section 4, 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 (26)) that it imposes fails. We examine an instance of this in Section 6.2, where a standard property of list reverse fails to carry over to the case of lists with interleaved effects.

6.1 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 $(\mu(ListF\ a|m), in_{ListF\ a}, in_m)$ 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:

$$
\begin{aligned}
&eAppend :: \mu(ListF\ a|m) \to \mu(ListF\ a|m) \to \mu(ListF\ a|m) \\
&eAppend\ xs\ ys = (\!|k|in_m|\!)\ xs \\
&\quad \textbf{where}\ k\ \ \ :: ListF\ a\ (\mu(ListF\ a|m)) \to \mu(ListF\ a|m) \\
&\qquad k\ \mathsf{Nil}\ \ \ = ys \\
&\qquad k\ (\mathsf{Cons}\ a\ xs) = in_{ListF\ a}\ (\mathsf{Cons}\ a\ xs)
\end{aligned}
$$

Note that, unlike the direct definition of *eAppend* that we made in Section 4, 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 $in_{ListF\ a}$. The fact that the pure part of the definition (i.e., the function k) is almost identical to the k 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*. We have one property for each of the constructors of the type constructor *ListF a*:

$$eAppend\ (in_{ListF\ a}\ \mathsf{Nil})\ ys = ys \tag{27}$$

$$eAppend\ (in_{ListF\ a}\ (\mathsf{Cons}\ a\ xs))\ ys = in_{ListF\ a}\ (\mathsf{Cons}\ a\ (eAppend\ xs\ ys)) \tag{28}$$

Both of these equations follow from the fact that the $(\!|k|in_m|\!)$ in the definition of *eAppend* is an f-and-m-algebra homomorphism, using Equation (23).

Again by construction, we also know that for any fixed ys, $\lambda xs.\ eAppend\ xs\ ys$ is an m-Eilenberg–Moore-algebra homomorphism. Hence we have the following property of *eAppend* for free, from Equation (24). For all $x :: m\ (\mu(ListF\ a|m))$:

$$eAppend\ (in_m\ x)\ ys = in_m\ (fmap_m\ (\lambda xs.\ eAppend\ xs\ ys)\ x) \tag{29}$$

If we unfold the definition of in_m, we can see that Equation (29) captures the fact that *eAppend* always evaluates its first argument. This is made clearer if we write in_m using the inverse function to the constructor List, $unList\ (\mathsf{List}\ xs) = xs$, yielding the following equation that is equivalent to Equation (29):

$$\begin{array}{rl} & eAppend\ (\mathsf{List}\ (\mathbf{do}\ \{xs \leftarrow x; unList\ xs\}))\ ys \\ = & \mathsf{List}\ (\mathbf{do}\ \{xs \leftarrow x; unList\ (eAppend\ xs\ ys)\}) \end{array}$$

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 $\lambda xs.\ eAppend\ xs\ ys$ is an m-Eilenberg–Moore-algebra homomorphism for all ys (Equation (29)).

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 Equations (25) and (26). Firstly, for all $x :: ListF\ a\ (\mu(ListF\ a|m))$, we must show that Equation (25) holds, i.e. that:

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

where

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

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 Equations (27) and (28).

Secondly, we must show that the right hand side of the equation to be proved is an m-Eilenberg–Moore-algebra homomorphism, i.e., that Equation (26) holds:

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

This follows straightforwardly from the fact that $\lambda xs.eAppend\ xs\ ys$ is itself an m-Eilenberg–Moore-algebra homomorphism for all ys, as we noted above in Equation (29), and that such homomorphisms are closed under composition:

$$\begin{array}{rl} & eAppend\ (eAppend\ (in_m\ x)\ ys)\ zs \\ = & \quad \{\text{Equation (29)}\} \\ & eAppend\ (in_m\ (fmap_m\ (\lambda xs.\ eAppend\ xs\ ys)\ x))\ zs \\ = & \quad \{\text{Equation (29)}\} \\ & in_m\ (fmap_m\ (\lambda xs.\ eAppend\ xs\ zs)\ (fmap_m\ (\lambda xs.\ eAppend\ xs\ ys)\ x)) \\ = & \quad \{fmap_m \text{ preserves function composition (Equation (2))}\} \\ & in_m\ (fmap_m\ (\lambda xs.\ eAppend\ (eAppend\ xs\ ys)\ zs)\ x) \end{array}$$

□

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 4. 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.

This proof, and the f-algebra proof in Section 4, are both generic in the monad m that we use to represent effects. In particular, if we instantiate m to be the non-termination monad, then we have proved that list append for Haskell's standard lazy lists is associative, without having to explicitly deal with a Cpo semantics.

6.2 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, we 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:

$$reverse\ (append\ xs\ ys) = append\ (reverse\ ys)\ (reverse\ xs) \tag{30}$$

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 the interleaved effects involve the possibility of non-termination, as in the $List_{lazy}$ example in Section 1.1, then *reverse* may never get to the last element of the list.

If we try to prove this property using Proof Principle 2, we see that we are unable to prove Equation (26), namely that the right hand side of the effectful version of Equation (30) (Equation (31), below) must be an *m*-Eilenberg–Moore-algebra homomorphism in the variable *xs*.

However, 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.

$$
\begin{array}{l}
eReverse :: \mu(ListF\ a|m) \to \mu(ListF\ a|m) \\
eReverse = (\!|k|in_m|\!) \\
\quad \textbf{where}\ k \qquad\qquad\quad :: ListF\ a\ (\mu(ListF\ a|m)) \to \mu(ListF\ a|m) \\
\qquad\quad k\ \mathsf{Nil} \qquad\quad = in_{ListF\ a}\ \mathsf{Nil} \\
\qquad\quad k\ (\mathsf{Cons}\ a\ xs) = eAppend\ xs\ (in_{ListF\ a}\ (\mathsf{Cons}\ a\ (in_{ListF\ a}\ \mathsf{Nil})))
\end{array}
$$

Verifying the effectful analogue of Equation (30) 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 $(\!|k'|in_m|\!)$, where

$$
\begin{array}{lcl}
k' :: ListF\ a\ (\mu(ListF\ a|m)) \to \mu(ListF\ a|m) & & \\
k'\ \mathsf{Nil} & = & eReverse\ ys \\
k'\ (\mathsf{Cons}\ a\ xs) & = & eAppend\ xs\ (in_{ListF\ a}\ (\mathsf{Cons}\ a\ (in_{ListF\ a}\ \mathsf{Nil})))
\end{array}
$$

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:

$$(\!|k'|in_m|\!)\ xs = eAppend\ (eReverse\ ys)\ (eReverse\ xs) \tag{31}$$

Then the pure part of the proof goes through straightforwardly. We are left with proving that $\lambda xs.\ eAppend\ (eReverse\ ys)\ (eReverse\ xs)$ (i.e., the right hand side of this equation) is an *m*-Eilenberg–Moore-algebra homomorphism for all *ys*. Certainly, *eReverse* is an *m*-Eilenberg–Moore-algebra homomorphism by its construction via the initial *f*-and-*m*-algebra property. However, $\lambda ys.\ eAppend\ xs\ ys$ is not an *m*-Eilenberg–Moore algebra homomorphism for all *xs*, as the following counterexample shows.

Let the monad *m* be the *ErrorM* monad we defined in Section 5.1. If *eAppend* were an *ErrorM*-Eilenberg–Moore-algebra homomorphism in its second argument the following equation would hold:

$$
\begin{array}{lcl}
 & & eAppend\ (in_{ListF\ a}\ (\mathsf{Cons}\ a\ (in_{ListF\ a}\ \mathsf{Nil})))\ (in_{ErrorM}\ (\mathsf{Error}\ \texttt{"msg"})) \\
= & & in_{ErrorM} \\
 & & \quad (fmap_{ErrorM}\ (eAppend\ (in_{ListF\ a}\ (\mathsf{Cons}\ a\ (in_{ListF\ a}\ \mathsf{Nil}))))) \\
 & & \qquad (\mathsf{Error}\ \texttt{"msg"}))
\end{array}
\tag{32}
$$

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

$$\begin{aligned} & eAppend\ (in_{ListF\ a}\ (\mathsf{Cons}\ a\ (in_{ListF\ a}\ \mathsf{Nil})))\ (in_{ErrorM}\ (\mathsf{Error}\ \texttt{"msg"})) \\ = & \qquad \{\text{Equation (28)}\} \\ & in_{ListF\ a}\ (Cons\ a\ (eAppend\ (in_{ListF\ a}\ \mathsf{Nil})\ (in_{ErrorM}\ (\mathsf{Error}\ \texttt{"msg"})))) \\ = & \qquad \{\text{Equation (27)}\} \\ & in_{ListF\ a}\ (Cons\ a\ (in_{ErrorM}\ (\mathsf{Error}\ \texttt{"msg"}))) \end{aligned}$$

while the right hand side of Equation (32) reduces by the definition of $fmap_{ErrorM}$ to simply:

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

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

7 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 any 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.

7.1 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 4 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 involve 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 $(\mu(f \circ m), in)$, then $m\ (\mu(f \circ m))$ is the carrier of an initial f-and-m-algebra.

Proof
See Appendix A. □

In Atkey, Ghani, Jacobs and Johann's 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.

7.2 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.

The seed of our construction is the existence of an initial $(f \circ m)$-algebra. Therefore, we need to first construct the composite functor $f \circ m$. To express the composition of two type operators as a new type operator, we introduce a **newtype**, as follows[4]:

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

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 an initial f-and-m-algebra by setting $\mu(f|m)$ to be the type $MuFM\ f\ m$, defined as:

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

Unfolding the definitions of $f :\circ: m$ and Mu shows that the type $MuFM\ f\ m$ is, up to isomorphism, the same as the type $MuFM_0\ f\ m$ from Section 1.1 that we arrived at by generalising the $List_{io}$ and $List_{lazy}$ examples.

The f-algebra and m-Eilenberg–Moore-algebra structure maps in_f and in_m are defined following the construction in Theorem 4:

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

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

Finally, we construct the unique f-and-m-algebra-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:

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

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

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

[4] This definition requires the GHC extension `-XTypeOperators`, allowing infix type constructors.

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 m-Eilenberg–Moore algebra. Unfortunately, it is not possible to express this requirement in Haskell's type system.

8 Application: Streaming I/O and coproducts of free monads with arbitrary monads

In Section 1.1, we motivated the consideration of streams of interleaved data and effects by giving the *hGetContents* function a type that more precisely reflects its actual behaviour. The Haskell community, motivated by the concerns about lazy I/O that we listed in Section 1.1, has proposed several other datatypes that capture the interleaving of effects with pure data[5], in order to make the interleaving explicit. One of the earliest was Kiselyov's iteratees (Kiselyov, 2012). Iteratees are used to support lazy I/O in languages such as Haskell by handling different kinds of sequential information processing in an incremental way.

Iteratees are descriptions of functions that alternate reading from some input with effects in some monad, eventually yielding some output. Kiselyov captured this using the following datatype, which follows the same pattern of mutual recursion as the *List m a* datatype declaration from Section 1.1:

```
data Reader' m a b                    newtype Reader m a b =
  = Input (Maybe a → Reader m a b)      Reader (m (Reader' m a b))
  | Yield b
```

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 of type a. 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, and notes that it has several useful properties, including the fact that it is (the functor part of) a monad. In terms of f-and-m-algebras we can see that the type *Reader m a b* is an initial f-and-m-algebra, where the functor f is given by:

```
data ReaderF a b x
  = Input (Maybe a → x)
  | Yield b
```

With this formulation, we could use Proof Principle 2 to reason about programs involving iteratees. For example, we could prove Kiselyov's result that *Reader m a b* is a monad whenever m is.

However, we can see iteratees as an instance of a yet more general construction: the coproduct of a free monad with an arbitrary monad m. Monad coproducts

[5] 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/).

provide a general and canonical way of specifying the combination of two monads (Lüth & Ghani, 2002) (we formally define coproducts of monads in Section 8.3, below). Almost trivially, once we observe that *Reader m a* is the coproduct of two monads, we can immediately deduce that it is a monad, rather than having to prove this fact as a special case for *Reader m a*. As we will see below in Section 8.4, the coproduct of a free monad and an arbitrary monad can be straightforwardly constructed using initial f-and-m-algebras. Much of this straightforwardness rests on the clear separation of pure and effectful concerns afforded by f-and-m-algebras.

Following on from Kiselyov's work, Gonzalez implemented the `pipes` library. The central definition of the `pipes` library is the *Proxy a′ a b′ b m r* datatype, which generalises Kiselyov's *Reader m a b* type. Gonzalez defines the *Proxy* type as follows:

```
data Proxy a' a b' b m r
  = Request a' (a → Proxy a' a b' b m r)
  | Respond b (b' → Proxy a' a b' b m r)
  | M (m (Proxy a' a b' b m r))
  | Pure r
```

In essence, a value of type *Proxy a′ a b′ b m r* is a tree of *requests* of type *a′*, reading values of type *a*, and *responses* of type *b*, reading values of type *b′*, interleaved with effects described by the monad *m*, finally yielding values of type *r*. Thus, Gonzalez's type adds the possibility of bidirectional requests and responses to Kiselyov's *Reader* type (hence the name "`pipes`" of the library).

Gonzalez proves several properties of the *Proxy* type constructor directly[6], including demonstrating that, just as was the case for Iteratees, it forms (the functor part of) a monad. Gonzalez's proofs appeal to an informal notion of coinduction in order to handle recursion in the presence of potentially infinite *Proxy a′ a b′ b m r* values arising from Haskell's non-strict semantics (recall the discussion in Section 1.1). Just as with Iteratees, we could observe that *Proxy* types are instances of data interleaved with effects, and reformulate Gonzalez's proofs using Proof Principle 2, accounting for non-strictness by treating it just as an arbitrary monad. However, we can again save work by observing that *Proxy a′ a b′ b m r* is an instance of the coproduct of a free monad and the monad *m*, and consequently the result that *Proxy a′ a b′ b m r* is a monad follows.

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 interaction trees that can be interpreted in multiple ways. In Section 8.2, 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 8.4. We present the formal definition of the monad coproduct in Section 8.3, and elaborate on the reading of free monads as interaction trees, now interleaved with effects from some arbitrary monad. In Section 8.4, we present a concrete construction of the coproduct of a free monad with an arbitrary monad. By using f-and-m-algebras we are able

[6] https://github.com/Gabriel439/Haskell-Pipes-Library/blob/master/laws.md

to reuse much of the core of the definitions of the free monad structure we defined in Section 8.2.

8.1 Free monads

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 interpretation. Swierstra & Altenkirch (2007) have developed this idea to provide a straightforward way of reasoning about programs that perform input/output. We will briefly describe this view of free monads after we give the formal definition:

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:

$$wrap_f \circ fmap_f\ (fmap_{FreeM\ f}\ g) = fmap_{FreeM\ f}\ g \circ wrap_f \tag{33}$$

$$wrap_f \circ fmap_f\ join_{FreeM\ f} = join_{FreeM\ f} \circ wrap_f \tag{34}$$

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 8.4 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 "interaction trees". 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 trees 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:

$$\begin{array}{l} read :: FreeM\ (ReaderF\ a)\ a \\ read = wrap_{ReaderF\ a}\ (\mathsf{Read}\ return_{FreeM\ (ReaderF\ a)}) \end{array}$$

Every free monad is a monad, so 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.

$$\begin{array}{l} swapRead :: FreeM\ (ReaderF\ String)\ (String, String) \\ swapRead = \textbf{do}\ \{s_1 \leftarrow read\,; s_2 \leftarrow read\,; return\ (s_2, s_1)\} \end{array}$$

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:

$$\begin{array}{l} useGetLine :: ReaderF\ String\ a \to IO\ a \\ useGetLine\ (\mathsf{Read}\ k) = \textbf{do}\ \{s \leftarrow getLine\,; return\ (k\ s)\} \end{array}$$

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

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

Applying $\langle\!\langle useGetLine \rangle\!\rangle$ 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 (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 effects in a monad m:

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

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 the interpretation of concrete effects. As we have observed in the case of list append in Section 4, 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 8.3 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.

8.2 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 μ(*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 8.4 we will be able to reuse many of the definitions in Figure 1.

8.3 Coproducts of monads

Monad coproducts provide a canonical way of describing the combination of two monads to form another monad. We can think of the coproduct of two monads

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 = (|k|)
  where k (Var a)  = in (Var (g a))
        k (Term x) = in (Term x)

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

join_FreeM f :: FreeM f (FreeM f a) → FreeM f b
join_FreeM f = (|j|)
  where j (Var x)  = x
        j (Term x) = in (Term x)

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

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

Fig. 1. Constructing free monads via f-algebras

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. 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*, as we shall see in Section 8.4, below.

Formally, 'least commitment' is realised as the existence of a *unique* monad morphism out of a coproduct for every way of interpreting its constituent 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 $(m_1, fmap_{m_1}, return_{m_1}, join_{m_1})$ and $(m_2, fmap_{m_1}, return_{m_2}, join_{m_2})$ be a pair of monads. A *coproduct* of these two monads is a monad $(m_1{+}m_2, fmap_{m_1+m_2}, return_{m_1+m_2}, join_{m_1+m_2})$ along with a pair of monad morphisms:

$$\begin{array}{lcl} inj_1 & :: & m_1\ a \to (m_1{+}m_2)\ a \\ inj_2 & :: & m_2\ a \to (m_1{+}m_2)\ a \end{array}$$

and the property that for any monad $(m, fmap_m, return_m, join_m)$ 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

$$\begin{array}{lcl} [g_1, g_2] \circ inj_1 & = & g_1 \\ [g_1, g_2] \circ inj_2 & = & g_2 \end{array}$$

In Section 8.1, we demonstrated how the free monad over a functor describing read commands allowed us to provide multiple interpretations of 'reading'. The monad coproduct $(FreeM\ (ReaderF\ String)){+}IO$ freely combines the abstract read commands described by the functor $ReaderF\ String$ with the concrete input/output actions of the IO monad. We view $(FreeM\ (ReaderF\ String)){+}IO$ as the modular reconstruction of the Iteratee monad $Reader\ m\ a$ we presented in Section 8.

The following example extends the *swapRead* example from Section 8.1 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:

```
swapRead2 :: ((FreeM (ReaderF String))+IO) ()
swapRead2 = do s1 ← inj1 read
               s2 ← inj1 read
               inj2 (putStrLn ("(" ++ s2 ++ "," ++ s1 ++ ")"))
```

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.

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 8.1:

```
[⟪useGetLine⟫, id] :: ((FreeM (ReaderF String))+IO) a → 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[7]:

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

[7] This functionality is very similar to the standard Scheme `with-input-from-file` function, which temporarily uses a file as the source for input, rather than the terminal.

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

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

Abstracting over the meaning of symbols such as *read* as we have done here is of course not new. The basic feature of the λ-calculus is to allow abstraction over the meaning of symbols. We could have written *swapRead2* as follows, using λ-abstractions rather than the monad coproduct:

```
swapRead2 :: IO String → IO ()
swapRead2 read = do s1 ← read
                    s2 ← read
                    putStrLn ("(" ++ s2 ++ "," ++ s1 ++ ")")
```

The two different interpretations above could then be obtained as *swapRead2 getLine* and *swapRead2* (*hGetLine h*). However, this approach becomes unwieldy if the definition of *swapRead2* becomes more complex: the parameter *read* needs to be passed through to all other functions that might need to do abstracted reading, and it is the responsibility of the programmer to do this plumbing manually. With the monad coproduct approach, the plumbing is handled automatically. Another advantage of monad coproducts over λ-abstraction of command interpretations is that we have access to the pure data constructors describing the commands. In the `pipes` library, for example, composition of *Proxy* values into a pipeline relies on being able to match the `Request` constructors of one *Proxy* with the `Response` constructors of another. Abstraction over opaque *IO* actions, as in the alternative *swapRead2* above, does not permit this kind of introspection.

8.4 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.

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 6: 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 (*FreeMF f a*)-and-m-algebra stucture – $in_{FreeMF\ f\ a}$ and in_m – for the first and second injections inj_1 and inj_2 respectively. Since $in_{FreeMF\ f\ a}$ injects an single abstract command from f into the coproduct, we use the free monad structure to inject all the commands into the coproduct.

$$
\begin{aligned}
&\textbf{type}\ ((\mathit{FreeM}\ f){+}m)\ a = \mu(\mathit{FreeMF}\ f\ a|m)\\
&\\
&\mathit{fmap}_{(\mathit{FreeM}\ f)+m} :: (a \to b) \to ((\mathit{FreeM}\ f)+m)\ a \to ((\mathit{FreeM}\ f)+m)\ b\\
&\mathit{fmap}_{(\mathit{FreeM}\ f)+m}\ g = (\!|k|\mathit{in}_m|\!)\\
&\quad \textbf{where}\ k\ (\mathsf{Var}\ a) \ = \mathit{in}_{\mathit{FreeMF}\ f\ b}\ (\mathsf{Var}\ (g\ a))\\
&\qquad\quad k\ (\mathsf{Term}\ x) = \mathit{in}_{\mathit{FreeMF}\ f\ b}\ (\mathsf{Term}\ x)\\
&\\
&\mathit{return}_{(\mathit{FreeM}\ f)+m} :: a \to ((\mathit{FreeM}\ f)+m)\ a\\
&\mathit{return}_{(\mathit{FreeM}\ f)+m}\ a = \mathit{in}_{\mathit{FreeMF}\ f\ a}\ (\mathsf{Var}\ a)\\
&\\
&\mathit{join}_{(\mathit{FreeM}\ f)+m} :: ((\mathit{FreeM}\ f)+m)\ (((\mathit{FreeM}\ f)+m)\ a) \to ((\mathit{FreeM}\ f)+m)\ a\\
&\mathit{join}_{(\mathit{FreeM}\ f)+m} = (\!|j|\mathit{in}_m|\!)\\
&\quad \textbf{where}\ j\ (\mathsf{Var}\ x) \ = x\\
&\qquad\quad j\ (\mathsf{Term}\ x) = \mathit{in}\ (\mathsf{Term}\ x)\\
&\\
&\mathit{inj}_1 :: \mathit{FreeM}\ f\ a \to ((\mathit{FreeM}\ f)+m)\ a\\
&\mathit{inj}_1 = (\!|\mathit{in}_{\mathit{FreeMF}\ f\ a} \circ \mathsf{Term}|\!)\\
&\\
&\mathit{inj}_2 :: m\ a \to ((\mathit{FreeM}\ f)+m)\ a\\
&\mathit{inj}_2 = \mathit{in}_m \circ \mathit{fmap}_m\ \mathit{return}_{(\mathit{FreeM}\ f)+m}\\
&\\
&[-,-] :: (\forall a.\ \mathit{FreeM}\ f\ a \to m'\ a) \to (\forall a.\ m\ a \to m'\ a) \to ((\mathit{FreeM}\ f)+m)\ a \to m'\ a\\
&[g_1, g_2] = (\!|c|\mathit{join}_{m'} \circ g_2|\!)\\
&\quad \textbf{where}\ c\ (\mathsf{Var}\ a) \ = \mathit{return}_{m'}\ a\\
&\qquad\quad c\ (\mathsf{Term}\ x) = \mathit{join}_{m'}\ ((\!|g_1|\!)^{-1}\ x)
\end{aligned}
$$

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

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 $(\!|c|\mathit{join}_{m'} \circ g_2|\!)$ is actually an m-Eilenberg–Moore-algebra. For the first law (Equation (20)), we reason as follows:

$$
\begin{aligned}
& \mathit{join}_{m'} \circ g_2 \circ \mathit{return}_m\\
= & \qquad\qquad \{g_2 \text{ is a monad morphism (Equation (11))}\}\\
& \mathit{join}_{m'} \circ \mathit{return}_{m'}\\
= & \qquad\qquad \{\text{monad law: } \mathit{join}_{m'} \circ \mathit{return}_{m'} = \mathit{id} \text{ (Equation (5))}\}\\
& \mathit{id}
\end{aligned}
$$

The second law (Equation (21)) is also straightforward:

$$
\begin{aligned}
& \mathit{join}_{m'} \circ g_2 \circ \mathit{join}_m\\
= & \qquad\qquad \{g_2 \text{ is a monad morphism (Equation (12))}\}\\
& \mathit{join}_{m'} \circ \mathit{join}_{m'} \circ g_2 \circ \mathit{fmap}_m\ g_2\\
= & \qquad\qquad \{\text{monad law: } \mathit{join}_{m'} \circ \mathit{join}_{m'} = \mathit{join}_{m'} \circ \mathit{fmap}'_m\ \mathit{join}_{m'} \text{ (Equation (7))}\}\\
& \mathit{join}_{m'} \circ \mathit{fmap}_{m'}\ \mathit{join}_{m'} \circ g_2 \circ \mathit{fmap}_m\ g_2\\
= & \qquad\qquad \{\text{naturality of } g_2\}\\
& \mathit{join}_{m'} \circ g_2 \circ \mathit{fmap}_m\ \mathit{join}_{m'} \circ \mathit{fmap}_m\ g_2\\
= & \qquad\qquad \{\mathit{fmap}_m \text{ preserves function composition (Equation (2))}\}\\
& \mathit{join}_{m'} \circ g_2 \circ \mathit{fmap}_m\ (\mathit{join}_{m'} \circ g_2)
\end{aligned}
$$

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

We emphasise that the result we have presented here is not new; Hyland *et al*. have already demonstrated, albeit with a different proof technique, that the construction we have given here 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 & Gibbons (2012). Our contribution is to show that the use of f-and-m-algebras simplifies and elucidates the definitions involved.

9 Conclusions

We have presented a generalisation of Filinski and Støvring's f-and-m-algebras to arbitrary categories, and seen how they simplify defining and reasoning about functions that manipulate interleaved data and effects. The key observation is that initial f-and-m-algebras are the analogue for the effectful setting of initial f-algebras in the pure setting. As such, they support the transporting of the standard definitional and proof principles to the effectful setting. This allows the implicit interleaving of data with effects, such as I/O and non-termination, to be made explicit and properly reflected in functions' types. Because they separate pure and effectful concerns, f-and-m-algebras support the direct transfer of definitions and proofs — as illustrated with our running example of list append — from the pure setting to the effectful setting. We have further shown how programming with initial f-and-m-algebras can be made practical by giving a generic construction of them in terms of $(f \circ m)$-algebras. Finally, we have argued that other datatypes that interleave data and effects in languages such as Haskell can be expressed as coproducts of free monads with arbitrary monads, and can thus be straightforwardly constructed using initial f-and-m-algebras.

9.1 Related work

The earliest attempt to incorporate effects into the initial algebra methodology appears to be Sheard's (1993a; 1993b) use of compile-time reflection to give direct constructions of monadic *map* and *fold* functions. Fokkinga (1994) and later Pardo (2004), generalised Sheard's constructions to the general categorical setting, giving a generic recursion combinator for effectful recursive computations that has type

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

and whose definition requires the existence of a *distributive law* $d :: f\ (m\ a) \to m\ (f\ a)$ describing how effects percolate through pure data in a uniform way. Fokkinga and Pardo both defined such distributive laws by induction over a grammar of regular functors, and then used them, together with liftings of functors to Kleisli categories (Barr & Wells, 1990; Mulry, 1995), to define monadic *fold*s for regular

datatypes. The result was an effectful structural recursion scheme over pure regular data in which all effects are pushed to the 'outside' to monadically wrap a pure result.

In fact, as both Fokkinga and Pardo show, the existence of a distributive law for just the binary product functor is all that is actually required to ensure that distributive laws exist for all regular functors. However, such laws need not exist for all monads; there is no distributive law for binary products for the state monad, for example. In any case, the assumption that distributive laws exist is too strong for the purposes of this article: we are concerned here with structural recursion over *effectful* data, in which data and effects are interleaved, rather than just monadically wrapped data.

Although Fokkinga and Pardo work in the same effectful setting, Pardo transfers more origami programming ideas from the pure setting to the effectful one than Fokkinga does. In addition to defining the aforementioned monadic *fold*s (catamorphisms), Pardo dualises them to give monadic *unfold*s (anamorphisms) for structuring corecursive programs with monadic effects. He also defines monadic hylomorphisms to support even more general ways of structuring monadic computations and combining their results. Interestingly, monadic hylomorphisms do achieve some interleaving of recursive calls to effectful computations with other computations, but the computations they structure must still consume pure data. Pardo also develops rules for fusing monadic programs structured using the monadic constructs he defines.

Meijer & Jeuring (1995) further extend ideas of origami programming to the effectful setting by developing a number of monadic fusion rules. Among these is a new short cut fusion rule for eliminating (pure) intermediate structures of type fa for regular functors f in a monadic context m. Jürgensen (2002) and Voigtländer (2008) also define monadic fusion rules based on the uniqueness of the map from a free monad to any other monad. Like the aforementioned recursion schemes, many methods based on initial algebras for restricted classes of (pure) datatypes are in fact generalisable to arbitrary inductive types. For example, Ghani & Johann (2009) give a short cut fusion rule that can eliminate data structures of any (pure) inductive type in any monadic context.

The work of Filinski & Støvring (2007) is undoubtedly the most closely related to ours. As we do in this article, they give *fold*s for datatypes with proper interleaving of effects. They do so first considering the case of lazy datatypes, and then generalising to datatypes that interleave monadic effects other than nontermination with pure data. To first model the way laziness interleaves the possibility of nontermination at any point in the production of a data structure, and then to model more general interleavings of effects, Filinski and Støvring work in the specific category Cpo, and with a specific grammar of what might be called "effectful regular functors" that allow effects in recursive positions. Their principle of *definition by rigid induction* amounts to the derivation of *fold*s for minimal invariants for monads in Cpo. A minimal invariant is a special case — in the specific setting of Cpo for the lifting monad m and an effectful regular functor f — of the carrier of the initial f-and-m-algebra, and Filinski and Støvring's *fold*s are special cases of our *fold*s from

Definition 11. Such monadic *fold*s differ from those of Fokkinga, Pardo, and Meijer and Jeuring in that they derive from initiality in the category of f-and-m-algebras, rather than from initiality of algebras in the Kleisli category for m under the auspices of a distributive law for f and m. Because initial f-and-m-algebras properly interleave effectful computations with the construction of pure data, so that effects are actually an integral part of the data being processed rather than just wrapping it, more general recursive patterns of effectful computation are possible.

Given the well-known relationship between *fold*s and induction, it is perhaps surprising that the papers preceding Filinski and Støvring's do not derive induction rules or other proof principles for effectful datatypes. Filinski and Støvring do, however, give a principle of *proof by rigid induction* for such datatypes that is a variant of those of both Lehmann & Smyth (1981) and Crole & Pitts (1992). More generally, their development supports the same kind of reasoning principles, again in the specific category and for the specific functors with which they work, that we show arbitrary initial f-and-m-algebras to support. The results reported in this article thus extend both the definitional principles of Filinksi and Støvring for structuring recursion over effectful datatypes, and their proof principles for reasoning about computations over such datatypes, to the general category-theoretic setting and to arbitrary functors. Filinski and Støvring also give fusion rules for effectful streams (although not for arbitrary effectful datatypes), and illustrate the extension of relational reasoning to effectful datatypes. We consider neither fusion rules nor relational reasoning here. Nevertheless, we see that this article generalises previous extensions of the initial algebra methodology to the effectful setting in three ways: it handles arbitrary functors, rather than special classes of functors; it handles actual interleaving of effects and data, rather than just the wrapping of pure results in effectful contexts; and it gives proof principles for reasoning about interleaved effectful computations, rather than just constructs for structuring those computations.

9.2 Future work

The monadic induction schemes supported by Filinski and Støvring, and generalised here, give one way to reason about effect-interleaved data. The 'fast and loose' reasoning advocated by Danielsson *et al.* (2006) is another. Using a logical relations style relation to relate total and non-total semantics of programs, Danielsson *et al.* show that programmers can reason about programs as though they were written in a total language and expect, in certain cases, to carry the results over to non-total languages. It would be useful from both practical and theoretical viewpoints to know if this kind of 'morally correct' reasoning can be extended to effects other than just nontermination. Indeed, while in this paper we have looked at reasoning principles that are valid for all monads m, including nontermination, it would be interesting to investigate what properties of pure data carry over to effectful data in the presence of monads with specific properties. For instance, the equation $eLength\ (eAppend\ xs\ ys) = eLength\ (eAppend\ ys\ xs)$, for a suitable definition of $eLength$, is not valid in the presence of effects described by an arbitrary monad m,

but is valid if m is *commutative*. By considering specific classes of monads, it may be possible to formulate specialised variants of Proof Principle 2.

In this article we have concentrated on demonstrating the utility of initial f-and-m-algebras for definition and reasoning, but we have not yet explored the potential for additional theoretical development of f-and-m-algebras. Fusion laws and other derived properties of initial f-and-m-algebras, extending the work of Filinski and Støvring that we mentioned above from streams to arbitrary interleaved data types, are the most obvious avenue for future work. A further line of future work lies in deeper investigation of the categorical properties of the category of f-and-m-algebras. In the present work, we constructed the category of f-and-m-algebras and showed that it had initial objects from first principles, while in Atkey *et al.*'s previous work (Atkey *et al.*, 2012), this was demonstrated by constructing an adjunction between the category of f-and-m-algebras and the category of $(f \circ m)$-algebras. An anonymous reviewer has pointed out the interesting property that the category of f-and-m-algebras is isomorphic to the category of $((\mathit{FreeM}\ f) + m)$-Eilenberg–Moore algebras, showing that the monad coproduct construction in Section 8 has a deeper significance. Further investigation of this kind of characterisation may lead to yet higher-level tools for defining and reasoning about programs that interleave pure data with effects.

Acknowledgements

We would like to gratefully acknowledge the work of Neil Ghani and Bart Jacobs, both on the original conference paper (Atkey *et al.*, 2012) that led to the work presented here, and on an earlier version of the present paper.

References

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

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

Barr, M. & Wells, C. (1990) *Category Theory for Computing Science*. Prentice Hall.

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

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

Crole, R. L. & Pitts, A. M. (1992) New foundations for fixpoint computations: Fix-hyperdoctrines and the fix-logic. *Inform. Comput.*, no. 52, 171–210.

Danielsson, N. A., Hughes, J., Jansson, P. & Gibbons, J. (2006) Fast and loose reasoning is morally correct. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, Charleston, South Carolina, USA: ACM, pp. 206–217.

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

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

Fokkinga, M. M. (1994) *Monadic Maps and Folds for Arbitrary Datatypes*. Technical Report Memoranda Informatica 94-28. University of Twente.

Ghani, N. & Johann, P. (2009) Short cut fusion of recursive programs with computational effects. In Proceedings of Trends in Functional Programming, TFP 2009, no. 9, Komarno, Solvakia: Kluwer, pp. 113–128.

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

Gibbons, J. (2003) Origami programming. In *The Fun of Programming*, Gibbons, J. & de Moor, O. (eds), 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. In Yeh, R. (ed), *Current Trends in Programming Methodology*, pp. 80–149.

Hyland, M., Plotkin, G. D. & Power, J. (2006) Combining effects: Sum and tensor. *Theor. Comput. Sci.* **357**(1-3), 70–99.

Jacobs, B. & Rutten, J. (2011) A tutorial on (co)algebras and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, Sangiorgi, D. & Rutten, J. (eds), Tracts in Theoretical Computer Science, no. 52. Cambridge University Press, pp. 38–99.

Jürgensen, C. (2002) *Using Monads to Fuse Recursive Programs (Extended Abstract)*. Available at: citeseer.ist.psu.edu/543861.html.

Kiselyov, O. (2012) Iteratees. In Schrijvers, T. & Thiemann, P. (eds), Proceedings of the 11th International Symposium on Functional and Logic Programming, FLOPS 2012, Lecture Notes in Computer Science, vol. 7294, Kanazawa, Japan: Springer, pp. 166–181.

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

Lehman, D. J. & Smyth, M. B. (1981) Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, no. 14, 97–139.

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

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

Meijer, E. & Jeuring, J. (1995) Merging moands and folds for functional programming. In *Proceedings of the First International Summer School on Advanced Functional Programming, AFP 1995*, vol. 925, Lecture Notes in Computer Science, Båstad, Sweden: Springer, pp. 228–266.

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

Mulry, P. S. (1995) Lifting theorems for Kleisli categories. In *Mathematical Foundations of Programming Semantics*, pp. 304–319.

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

Peyton Jones, S. L. & Wadler, P. (1993) Imperative functional programming. In Deusen, M. S. Van, & Lang, B. (eds), Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium

on Principles of Programming Languages, POPL 1993, Charleston, South Carolina, USA: ACM, pp. 71–84.

Piróg, M. & Gibbons, J. (2012) Tracing monadic computations and representing effects. In Chapman, J. & Levy, P. B. (eds), Proceedings 4th Workshop on Mathematically Structured Functional Programming, MSFP 2012, Electronic Proceedings in Theoretical Computer Science, vol. 76, Tallinn, Estonia: British Computer Society, pp. 90–111.

Pitts, A. M. (1996) Relational properties of domains. *Inf. Comput.* **127**(2), 66–90.

Sheard, T. (1993a) Adding algebraic methods to traditional functional languages by using reflection. *Algebraic Methods and Software Technology*.

Sheard, T. (1993b) *Type Parametric Programming with Compile-Time Reflection*. Technical Report, Oregon Graduate Institute of Science and Technology.

Sheard, T. & Pasalic, E. (2004) Two-level types and parameterized modules. *J. Funct. Program.* **14**(5), 547–587.

Swierstra, W. (2008) Data types à la carte. *J. Funct. Program.* **18**(4), 423–436.

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

Voightländer, J. (2008) Asymptotic improvement of computations over free monads. In *Proceedings of the 9th International Conference on Mathematics of Program Construction, MPC 2008*, Luminy, France: Springer, pp. 388–403.

Wadler, P., Taha, W. & MacQueen, D. 1998 (September) How to add laziness to a strict language without even being odd. *Proceedings of the ACM SIGPLAN Workshop on Standard ML*, Baltimore, Maryland, USA: ACM.

A Proof of Theorem 4

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 $(\mu(f \circ m), in)$, 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 in and the structure of the monad m. For the f-algebra component, we use the composite:

$$in_f = return_m \circ in :: f\ (m\ (\mu(f \circ m))) \to m\ (\mu(f \circ m))$$

The m-Eilenberg–Moore-algebra component is straightforward, using the free m-Eilenberg–Moore-algebra construction from Section 5.1:

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

Since we have used the free m-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, k, l). We construct, and prove unique, an f-and-m-algebra homomorphism h from the algebra $(m\ (\mu(f \circ m)), in_f, in_m)$ to the algebra with carrier a using the initiality of $\mu(f \circ m)$:

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

Close inspection of h reveals that it has the same structure as the definition of *eAppend* in terms of initial f-algebras we gave at the start of Section 4, where $l = join_m$. Therefore, as we noted in the introduction to Section 7.1, the construction we are building 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:

$$\begin{array}{ll} & h \circ in_f \\ = & \qquad \{\text{definitions of } h \text{ and } in_f\} \\ & l \circ fmap_m\ (\!|k \circ fmap_f\ l|\!) \circ return_m \circ in \\ = & \qquad \{\text{naturality of } return_m \text{ (Equation (8))}\} \\ & l \circ return_m \circ (\!|k \circ fmap_f\ l|\!) \circ in \\ = & \qquad \{l \text{ is an } m\text{-Eilenberg–Moore-algebra (Equation (20))}\} \\ & (\!|k \circ fmap_f\ l|\!) \circ in \\ = & \qquad \{(\!|-|\!) \text{ is an } (f \circ m)\text{-algebra homomorphism (Equation (3))}\} \\ & k \circ fmap_f\ l \circ fmap_f\ (fmap_m\ (\!|k \circ fmap_f\ l|\!)) \\ = & \qquad \{fmap_f \text{ preserves function composition (Equation (2))}\} \\ & k \circ fmap_f\ (l \circ fmap_m\ (\!|k \circ fmap_f\ l|\!)) \\ = & \qquad \{\text{definition of } h\} \\ & k \circ fmap_f\ h \end{array}$$

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

$$\begin{array}{ll} & h \circ in_m \\ = & \qquad \{\text{definitions of } h \text{ and } in_m\} \\ & l \circ fmap_m\ (\!|k \circ fmap_f\ l|\!) \circ join_m \\ = & \qquad \{\text{naturality of } join_m \text{ (Equation (9))}\} \\ & l \circ join_m \circ fmap_m\ (fmap_m\ (\!|k \circ fmap_f\ l|\!)) \\ = & \qquad \{l \text{ is an } m\text{-Eilenberg–Moore algebra (Equation (21))}\} \\ & l \circ fmap_m\ l \circ fmap_m\ (fmap_m\ (\!|k \circ fmap_f\ l|\!)) \\ = & \qquad \{fmap_m \text{ preserves function composition (Equation (2))}\} \\ & l \circ fmap_m\ (l \circ fmap_m\ (\!|k \circ fmap_f\ l|\!)) \\ = & \qquad \{\text{definition of } h\} \\ & l \circ fmap_m\ h \end{array}$$

3. The function h is the unique such f-and-m-algebra homomorphism. 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 $(\mu(f \circ m), in)$ to $(a, k \circ fmap_f\ l)$, as verified by the following steps:

$$
\begin{array}{ll}
 & h' \circ return_m \circ in \\
= & \quad \{\text{definition of } in_f\} \\
 & h' \circ in_f \\
= & \quad \{h' \text{ is an } f\text{-and-}m\text{-algebra homomorphism}\} \\
 & k \circ fmap_f\ h' \\
= & \quad \{\text{monad law: } join_m \circ fmap_m\ return_m = id \text{ (Equation (6))}\} \\
 & k \circ fmap_f\ (h' \circ join_m \circ fmap_m\ return_m) \\
= & \quad \{h' \text{ is an } m\text{-Eilenberg–Moore-algebra homomorphism (Equation (22))}\} \\
 & k \circ fmap_f\ (l \circ fmap_m\ h' \circ fmap_m\ return_m) \\
= & \quad \{fmap_f \text{ preserves function composition (Equation (2))}\} \\
 & k \circ fmap_f\ l \circ fmap_f\ (fmap_m\ (h' \circ return_m))
\end{array}
$$

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

$$h' \circ return_m = (\!|k \circ fmap_f\ l|\!) \tag{A 1}$$

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

$$
\begin{array}{ll}
 & h \\
= & \quad \{\text{definition of } h\} \\
 & l \circ fmap_m\ (\!|k \circ fmap_f\ l|\!) \\
= & \quad \{\text{Equation (A 1)}\} \\
 & l \circ fmap_m\ (h' \circ return_m) \\
= & \quad \{fmap_m \text{ preserves function composition (Equation (2))}\} \\
 & l \circ fmap_m\ h' \circ fmap_m\ return_m \\
= & \quad \{h' \text{ is an } m\text{-Eilenberg–Moore-algebra homomorphism (Equation (22))}\} \\
 & h' \circ join_m \circ fmap_m\ return_m \\
= & \quad \{\text{monad law: } join_m \circ fmap_m\ return_m = id\} \\
 & h'
\end{array}
$$

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

□