Interleaving Data and Effects

Patricia Johann
Appalachian State University
cs.appstate.edu/~johannp

Joint work with Bob Atkey, Neil Ghani, and Bart Jacobs

Haskell Symposium 2014
Programs, Data, and Effects

- Programming languages provide a wide array of constructs for storing and manipulating data
  - built-in data types (Bool, Int, Float,...)
  - lists
  - trees
  - arrays
Programs, Data, and Effects

- Programming languages provide a wide array of constructs for storing and manipulating data
  - built-in data types (Bool, Int, Float,...)
  - lists
  - trees
  - arrays
- Often these data types are pure, i.e., do not incorporate effects
Programs, Data, and Effects

• Programming languages provide a wide array of constructs for storing and manipulating data
  – built-in data types (Bool, Int, Float, ...)
  – lists
  – trees
  – arrays

• Often these data types are pure, i.e., do not incorporate effects

• However, sometimes data types not only to incorporate effects, but also to interleave them with pure data
Programming languages provide a wide array of constructs for storing and manipulating data

- built-in data types (Bool, Int, Float, ...)
- lists
- trees
- arrays

Often these data types are pure, i.e., do not incorporate effects.

However, sometimes data types not only to incorporate effects, but also to *interleave* them with pure data.

Unfortunately, this is *not always reflected in the types themselves*.
Scenario I: (Implicitly) Interleaved Non-termination

- Effects are implicitly built into every Haskell type: every Haskell type allows the possibility of non-termination while inspecting a pure value of that type
Scenario I: ( Implicitly) Interleaved Non-termination

- Effects are implicitly built into every Haskell type: every Haskell type allows the possibility of non-termination while inspecting a pure value of that type
- So not only is non-termination present in a type like [a], but because non-termination is possible at every Haskell type — including the element type a — it’s actually interleaved throughout the entire type!
Scenario I: (Implicitly) Interleaved Non-termination

- Effects are implicitly built into every Haskell type: every Haskell type allows the possibility of non-termination while inspecting a pure value of that type.
- So not only is non-termination present in a type like `[a]`, but because non-termination is possible at every Haskell type — including the element type `a` — it’s actually interleaved throughout the entire type!
- In particular, because of Haskell’s lazy semantics, Haskell data structures can be infinite, as well as finite.
  - `[a]` is the type of finite and infinite lists of elements of type `a`. 
**Scenario I: (Implicitly) Interleaved Non-termination**

- Effects are implicitly built into every Haskell type: every Haskell type allows the possibility of non-termination while inspecting a pure value of that type.

- So not only is non-termination present in a type like `[a]`, but because non-termination is possible at every Haskell type — including the element type `a` — it’s actually interleaved throughout the entire type!

- In particular, because of Haskell’s lazy semantics, Haskell data structures can be infinite, as well as finite.
  - `[a]` is the type of finite and infinite lists of elements of type `a`.

- But neither the presence of non-termination effects, nor their interleaving, is evident from the types themselves.
Scenario II: (Implicitly) Interleaved $IO$ Effects

- The type of the Haskell library function

\[ hGetContents :: Handle \rightarrow IO [Char] \]

suggests that it reads all the available data from the file referenced by $Handle$ as an $IO$ action and yields the list of characters as pure data.
Scenario II: (Implicitly) Interleaved \( IO \) Effects

- The type of the Haskell library function

\[
hGetContents :: \text{Handle} \rightarrow \text{IO} \left[ \text{Char} \right]
\]

suggests that it reads all the available data from the file referenced by \( \text{Handle} \) as an \( IO \) action and yields the list of characters as pure data.

- The standard implementation does not read data from the handle until the list is accessed by the program, so the effect of reading from the file handle is implicitly interleaved with computation on the (pure) list.
**Scenario II: (Implicitly) Interleaved IO Effects**

- The type of the Haskell library function
  
  $$hGetContents :: Handle \rightarrow IO [\text{Char}]$$

  suggests that it reads all the available data from the file referenced by
  *Handle* as an *IO* action and yields the list of characters as pure data.

- The standard implementation does not read data from the handle until
  the list is accessed by the program, so the effect of reading from the
  file handle is *implicitly interleaved* with computation on the (pure) list.

- This interleaving is **not reflected in the type of hGetContents**, so
  
  - *IO* errors that occur during reading are reported by throwing exceptions from pure code — possibly long after the call to *hGetContents*.
  - The handle is implicitly closed when the end of the file is reached, but if the end of file is never reached the handle will never be closed.
  - Since the programmer cannot always predict when reads will occur, it is not safe for them to close the file handle.
Question I:
How can we make the interleaving of data and effects explicit in types?
Inductive Data Types with Effects

- The type of lists interleaved with possible non-termination can be given as

\[
\text{data } \text{List}'_{\text{lazy}} a \\
= \text{Nil}_{\text{lazy}} \\
\mid \text{Cons}_{\text{lazy}} a (\text{List}_{\text{lazy}} a)
\]

\begin{align*}
\text{newtype } \text{List}_{\text{lazy}} a &= \\
&= \text{List}_{\text{lazy}} (\text{List}'_{\text{lazy}} a) \perp
\end{align*}
Inductive Data Types with Effects

- The type of lists interleaved with possible non-termination can be given as

\[
\text{data } List'_{\text{lazy}} a = \quad \text{newtype } List_{\text{lazy}} a = \\
= \text{Nil}_{\text{lazy}} \\
\mid \text{Cons}_{\text{lazy}} a \ (List_{\text{lazy}} a)
\]

- The type of lists interleaved with IO operations can be given as

\[
\text{data } List'_{\text{io}} = \quad \text{newtype } List_{\text{io}} = \\
= \text{Nil}_{\text{io}} \\
\mid \text{Cons}_{\text{io}} \text{ Char List}_{\text{io}} \\
\mid \text{List}_{\text{io}} (\text{IO List}'_{\text{io}})
\]
Question II:
How can we program effectively with, and reason effectively about, such “effectful” data types?
Structure of This Talk

- **Recall**: Standard initial algebra techniques
Structure of This Talk

- Recall: Standard initial algebra techniques
- Argue: Straightforward application of initial algebra techniques is at the wrong level of abstraction for effectful data types
Structure of This Talk

- **Recall**: Standard initial algebra techniques
- **Argue**: Straightforward application of initial algebra techniques is at the wrong level of abstraction for effectful data types
- **Instead**: Separate pure and effectful parts using \( f\text{-}and\text{-}m\text{-}algebras \)
Structure of This Talk

- **Recall**: Standard initial algebra techniques
- **Argue**: Straightforward application of initial algebra techniques is at the wrong level of abstraction for effectful data types
- **Instead**: Separate pure and effectful parts using *f-and-m-algebras*
  - *f*-algebras for a functor *f* describe the pure parts of an effectful data type
Structure of This Talk

- **Recall**: Standard initial algebra techniques
- **Argue**: Straightforward application of initial algebra techniques is at the wrong level of abstraction for effectful data types
- **Instead**: Separate pure and effectful parts using \( f \)-and-\( m \)-algebras
  - \( f \)-algebras for a functor \( f \) describe the pure parts of an effectful data type
  - \( m \)-Eilenberg-Moore algebras for a monad \( m \) describe the effects
Structure of This Talk

- **Recall**: Standard initial algebra techniques
- **Argue**: Straightforward application of initial algebra techniques is at the wrong level of abstraction for effectful data types
- **Instead**: Separate pure and effectful parts using \( f \)-and-\( m \)-algebras
  - \( f \)-algebras for a functor \( f \) describe the pure parts of an effectful data type
  - \( m \)-Eilenberg-Moore algebras for a monad \( m \) describe the effects
- **Represent**: Effectful data types as initial \( f \)-and-\( m \)-algebras
Structure of This Talk

- **Recall**: Standard initial algebra techniques
- **Argue**: Straightforward application of initial algebra techniques is at the wrong level of abstraction for effectful data types
- **Instead**: Separate pure and effectful parts using *f*-and-*m*-algebras
  - *f*-algebras for a functor *f* describe the pure parts of an effectful data type
  - *m*-Eilenberg-Moore algebras for a monad *m* describe the effects
- **Represent**: Effectful data types as initial *f*-and-*m*-algebras
- **Show**: Initial *f*-and-*m*-algebra techniques are at the right level of abstraction for effectful data types
Structure of This Talk

- **Recall**: Standard initial algebra techniques
- **Argue**: Straightforward application of initial algebra techniques is at the wrong level of abstraction for effectful data types
- **Instead**: Separate pure and effectful parts using *f*-and-*m*-algebras
  - *f*-algebras for a functor *f* describe the pure parts of an effectful data type
  - *m*-Eilenberg-Moore algebras for a monad *m* describe the effects
- **Represent**: Effectful data types as initial *f*-and-*m*-algebras
- **Show**: Initial *f*-and-*m*-algebra techniques are at the right level of abstraction for effectful data types
- **Revisit**: Motivating examples with initial *f*-and-*m*-algebra techniques
Initial Algebras for Pure Data Types (I)

- Model the individual “layers” of a data type using a functor

\[(f, \text{fmap} :: (a \to b) \to f\ a \to f\ b)\]

Here, \text{fmap} is assumed to preserve identities and composition
Model the individual “layers” of a data type using a functor

\[(f, \text{fmap} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b)\]

Here, \text{fmap} is assumed to preserve identities and composition

Describe how to reduce each “layer” in an inductive data structure to a value using an \(f\)-algebra

\[(a, k :: f\ a \rightarrow a)\]
Initial Algebras for Pure Data Types (I)

• Model the individual “layers” of a data type using a functor

\[(f, fmap :: (a \to b) \to f a \to f b)\]

Here, \(fmap\) is assumed to preserve identities and composition

• Describe how to reduce each “layer” in an inductive data structure to a value using an \(f\)-algebra

\[(a, k :: f a \to a)\]

• Characterize the data type as the carrier \(\mu f\) of the initial \(f\)-algebra

\[(\mu f, in : f(\mu f) \to \mu f)\]
An \( f \)-algebra homomorphism from an \( f \)-algebra \((a, k_a)\) to an \( f \)-algebra \((b, k_b)\) is a function \( h : a \to b \) such that

\[
\begin{align*}
  f &\xrightarrow{f \text{map} \, h} f \\
  k_a &\downarrow \quad \quad \quad \quad \quad \quad \quad \downarrow k_b \\
  a &\xrightarrow{h} b
\end{align*}
\]
Initial Algebras for Pure Data Types (II)

- An $f$-algebra homomorphism from an $f$-algebra $(a, k_a)$ to an $f$-algebra $(b, k_b)$ is a function $h :: a \rightarrow b$ such that

\[
\begin{array}{ccc}
fa & \xrightarrow{f \text{map } h} & fb \\
\downarrow{k_a} & & \downarrow{k_b} \\
\downarrow{h} & & \downarrow{\downarrow{}} \\
a & \rightarrow{h} & b
\end{array}
\]

- For every $f$-algebra $(a, k)$, there is a unique $f$-algebra homomorphism from the initial $f$-algebra $(\mu f, \text{in})$ to $(a, k)$

\[
\begin{array}{ccc}
f(\mu f) & \xrightarrow{f \text{map } (\mid k \mid)} & fa \\
\downarrow{\text{in}} & & \downarrow{k} \\
\mu f & \xrightarrow{(\mid k \mid)} & a
\end{array}
\]
• An $f$-algebra homomorphism from an $f$-algebra $(a, k_a)$ to an $f$-algebra $(b, k_b)$ is a function $h :: a \to b$ such that

\[
\begin{array}{c}
f \ a \xrightarrow{f \text{map} \ h} \ f \ b \\
k_a \downarrow \quad \quad \quad \quad \quad \quad \downarrow k_b \\
a \xrightarrow{h} \ b
\end{array}
\]

• For every $f$-algebra $(a, k)$, there is a unique $f$-algebra homomorphism from the initial $f$-algebra $(\mu f, \text{in})$ to $(a, k)$

\[
\begin{array}{c}
f(\mu f) \xrightarrow{f \text{map} \ (|k|)} f \ a \\
\downarrow \text{in} \quad \quad \quad \quad \quad \downarrow k \\
\mu f \xrightarrow{|k|} a
\end{array}
\]

• We denote the unique function from $\mu f$ to $a$ by $(|k|)$
The functor $ListF a$ describes the individual “layers” of a list

\[
\text{data } ListF \ a \ x \\
\begin{align*}
\text{Nil} & : \ (x \to y) \to ListF \ a \ x \\
\text{Cons} \ a \ x & :\ (x \to y) \to ListF \ a \ x \\
\end{align*}
\]

\[
\begin{align*}
\text{fmap} \ g \ \text{Nil} & = \text{Nil} \\
\text{fmap} \ g \ (\text{Cons} \ a \ xs) & = \text{Cons} \ a \ (g \ xs)
\end{align*}
\]
Example I — Initial Algebras for Lists

• The functor $ListF a$ describes the individual “layers” of a list

\[
\text{data } ListF a x = \begin{cases} 
\text{Nil} & \text{fmap } g \text{ Nil } = \text{Nil} \\
\text{Cons } a x & \text{fmap } g (\text{Cons } a x s) = \text{Cons } a (g x s)
\end{cases}
\]

• The type $[a]$ of finite lists is the carrier of the initial $(ListF a)$-algebra with

\[
\begin{align*}
in & : ListF a [a] \rightarrow [a] \\
in \text{ Nil } & = [] \\
in (\text{Cons } a x s) & = a : x s
\end{align*}
\]
Example I — Initial Algebras for Lists

- The functor $\text{ListF} a$ describes the individual “layers” of a list
  
  data $\text{ListF} a x$  
  
  $\text{Nil} = \text{Nil}$  
  
  $\text{Cons} a x$  

  fmap :: $(x \rightarrow y) \rightarrow \text{ListF} a x \rightarrow \text{ListF} a y$  

  $\text{fmap} g \text{Nil} = \text{Nil}$  

  $\text{fmap} g \text{(Cons} a x) = \text{Cons} a (g x)$

- The type $[a]$ of finite lists is the carrier of the initial $(\text{ListF} a)$-algebra with

  in :: $\text{ListF} a [a] \rightarrow [a]$  

  $\text{in} \text{Nil} = []$  

  $\text{in} \text{(Cons} a x) = a : xs$

- The fold for $[a]$ is

  $(| - |) :: (\text{ListF} a b \rightarrow b) \rightarrow [a] \rightarrow b$  

  $(|k|) [] = k \text{Nil}$  

  $(|k|) (a : xs) = k (\text{Cons} a (|k| xs))$
Example II — Initial Algebras Generically

- The carrier of the initial $f$-algebra for a functor $(f, fmap)$ can be implemented as

\[
data Mu f = \text{In} \{ \text{unIn} :: f (Mu f) \}\]
Example II — Initial Algebras Generically

- The carrier of the initial $f$-algebra for a functor $(f, fmap)$ can be implemented as

  \[
  \text{data } \mu f = \text{In } \{ \text{unIn} :: f (\mu f) \}\]

- The type $\mu f$ is the carrier of the initial $f$-algebra with

  \[
  \text{in} :: f (\mu f) \rightarrow \mu f
  \]

  \[
  \text{in} = \text{In}
  \]
The carrier of the initial $f$-algebra for a functor $(f, fmap)$ can be implemented as

```haskell
data Mu f = In { unIn :: f (Mu f) }
```

The type $Mu f$ is the carrier of the initial $f$-algebra with $in :: f (Mu f) \to Mu f$

```
in = In
```

The fold for $Mu f$ can be defined as

```
(| − |) :: Functor f ⇒ (f a → a) → Mu f → a

|k| = k \circ fmap (|k|) \circ unIn
```
What Have We Gained?

- *Definitional principles* for defining functions on data types
  - *fold* operators for expressing recursive functions
  - definition by pattern matching
What Have We Gained?

- **Definitional principles** for defining functions on data types
  - *fold* operators for expressing recursive functions
  - definition by pattern matching
- **Proof principles** for reasoning about such functions
  - induction rules
  - *fold* fusion rules
What Have We Gained?

- **Definitional principles** for defining functions on data types
  - *fold* operators for expressing recursive functions
  - definition by pattern matching

- **Proof principles** for reasoning about such functions
  - *fold* fusion rules
  - induction rules

- **Other tools for structured programming and reasoning** — e.g., introduction and elimination rules, computation (i.e., \( \beta \), from weak initiality) rules and extensionality (i.e., \( \eta \), from uniqueness) rules for *folds*; *build* combinators; *fold/build* rules...
What Have We Gained?

- **Definitional principles** for defining functions on data types
  - `fold` operators for expressing recursive functions
  - definition by pattern matching
- **Proof principles** for reasoning about such functions
  - `fold` fusion rules
  - induction rules
- **Other tools for structured programming and reasoning** — e.g., introduction and elimination rules, computation (i.e., β, from weak initiality) rules and extensionality (i.e., η, from uniqueness) rules for `folds`; `build` combinators; `fold/build` rules...

Above all, initial algebra semantics gives a **principled** approach to programming with data types that is **generic** over data types
**Exploiting Initiality**

**Proof Principle 1** Let \((a, k)\) be an \(f\)-algebra and \(g : \mu f \to a\) be a function. The equation

\[
(\lvert k \rvert) = g
\]

holds iff \(g\) is an \(f\)-algebra homomorphism, i.e., iff

\[
g \circ \text{in} = k \circ \text{fmap } g
\]
Representing \textit{append}

- Assume $(\mu(ListF a), \text{in})$ exists
Representing \textit{append}

- Assume \((\mu(ListF a), in)\) exists

- We can define \textit{append} in terms of \textit{fold} as

\[
\text{append} :: \mu(ListF a) \rightarrow \mu(ListF a) \rightarrow \mu(ListF a)
\]

\[
\text{append } xs \ ys = (|k|) \ xs
\]

where

\[
\begin{align*}
    k \text{ Nil} & = ys \\
    k \text{ (Cons } a \text{ xs)} & = in \text{ (Cons } a \text{ xs)}
\end{align*}
\]
Representing **append**

- Assume \((\mu(ListF \ a), \text{in})\) exists
- We can define **append** in terms of **fold** as

\[
\text{append} :: \mu(ListF \ a) \rightarrow \mu(ListF \ a) \rightarrow \mu(ListF \ a)
\]

\[
\text{append} \ x \ y \ = \ (|k|) \ x
\]

where \(k \ \text{Nil} \ = \ y\)

\[
k \ (\text{Cons} \ a \ x) \ = \ \text{in} \ (\text{Cons} \ a \ x)
\]

- Unfolding this definition gives these **equational properties** of **append**

\[
\text{append} \ (\text{in} \ \text{Nil}) \ y \ = \ y
\]

\[
\text{append} \ (\text{in} \ (\text{Cons} \ a \ x)) \ y \ = \ \text{in} \ (\text{Cons} \ a \ (\text{append} \ x \ y))
\]
Theorem: For all $xs, ys, zs :: \mu(ListF a)$,

\[ append \, xs \, (append \, ys \, zs) = append \, (append \, xs \, ys) \, zs \]
**Theorem:** For all $xs, ys, zs :: \mu(ListF a)$,

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

**Proof:**

1. Instantiate Proof Principle 1 and prove the equation

$$\langle k \rangle \; xs = append \; (append \; xs \; ys) \; zs$$
Theorem: For all $xs, ys, zs :: \mu(\text{ListF } a)$,

$$\text{append } xs \ (\text{append } ys \ zs) = \text{append } (\text{append } xs \ ys) \ zs$$

Proof:

1. Instantiate Proof Principle 1 and prove the equation

   $$\langle k \rangle \ xs = \text{append } (\text{append } xs \ ys) \ zs$$

   i.e.,

   $$\langle k \rangle = g$$

   where

   $$g = \lambda xs. \text{append } (\text{append } xs \ ys) \ zs$$

   $$k \ \text{Nil} = \text{append } ys \ zs$$

   $$k \ (\text{Cons } a \ xs) = \text{in } (\text{Cons } a \ xs)$$
2. It suffices to prove that

\[ g \circ \text{in} = k \circ \text{fmap } g \]

i.e., that for all \( x :: \text{ListF } a (\mu(\text{ListF } a)) \),

\[ = \text{append } (\text{append } (\text{in } x) \text{ ys}) \text{ zs} \]
\[ = k (\text{fmap } (\lambda \text{xs}. \text{append } (\text{append } \text{xs } \text{ ys}) \text{ zs}) x) \]
2. It suffices to prove that

\[ g \circ \text{in} = k \circ \text{fmap} \, g \]

i.e., that for all \( x :: \text{ListF} \, a (\mu (\text{ListF} \, a)) \),

\[ = \text{append} (\text{append} (\text{in} \, x) \, ys) \, zs \]

\[ = k (\text{fmap} (\lambda xs. \text{append} (\text{append} \, xs \, ys) \, zs) \, x) \]

3. Use case analysis according as \( x = \text{Nil} \) or \( x = \text{Cons} \, a \, xs \)
Associativity of \textit{append} (II)

2. It suffices to prove that

\[ g \circ \text{in} = k \circ \text{fmap} \ g \]

i.e., that for all \( x :: \text{ListF} \ a \ (\mu(\text{ListF} \ a)) \),

\[ = \ \text{append} \ (\text{append} \ (\text{in} \ x) \ ys) \ zs \]
\[ = \ k \ (\text{fmap} \ (\lambda \ xs. \ \text{append} \ (\text{append} \ xs \ ys) \ zs) \ x) \]

3. Use case analysis according as \( x = \text{Nil} \) or \( x = \text{Cons} \ a \ xs \)

4. For each case, we directly use the equational properties of \textit{append} and the definitions of \( g \) and \( \text{fmap} \) for \( \text{ListF} \ a \)
Associativity of \textit{append} (II)

2. It suffices to prove that

\[ g \circ \text{in} = k \circ \text{fmap} \ g \]

i.e., that for all \( x :: \text{ListF} \ a (\mu(\text{ListF} \ a)), \)

\[ = \ append (append (\text{in} x) \ ys) \ zs \]
\[ = \ k (\text{fmap} (\lambda xs. \ append (append xs \ ys) \ zs) \ x) \]

3. Use case analysis according as \( x = \text{Nil} \) or \( x = \text{Cons} \ a \ xs \)

4. For each case, we directly use the equational properties of \textit{append} and
   the definitions of \( g \) and \textit{fmap} for \textit{ListF} \ a

The proof is \textit{straightforward, easy, and short} (9 lines)
Monads for Effects

- Model an effect using a monad

\[(m, \text{fmap}_m, \text{return}_m, \text{join}_m)\]

where

\[\text{fmap}_m :: (a \rightarrow b) \rightarrow m a \rightarrow m b\]

\[\text{return}_m :: a \rightarrow m a\]

\[\text{join}_m :: m (m a) \rightarrow m a\]
Monads for Effects

• Model an effect using a monad

\((m, \text{fmap}_m, \text{return}_m, \text{join}_m)\)

where

\(\text{fmap}_m :: (a \rightarrow b) \rightarrow m a \rightarrow m b\)

\(\text{return}_m :: a \rightarrow m a\)

\(\text{join}_m :: m (m a) \rightarrow m a\)

• The monad laws must be satisfied
Monads for Effects

- Model an effect using a monad

\[(m, fmap_m, return_m, join_m)\]

where

\[fmap_m :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b\]

\[return_m :: a \rightarrow m\ a\]

\[join_m :: m\ (m\ a) \rightarrow m\ a\]

- The monad laws must be satisfied
- The naturality laws for \(return_m\) and \(join_m\) must be satisfied
Monads for Effects

- Model an effect using a monad
  
  \[(m, fmap_m, return_m, join_m)\]

  where

  \[fmap_m :: (a \rightarrow b) \rightarrow m a \rightarrow m b\]

  \[return_m :: a \rightarrow m a\]

  \[join_m :: m (m a) \rightarrow m a\]

- The monad laws must be satisfied

- The naturality laws for \(return_m\) and \(join_m\) must be satisfied

- Examples are the non-termination monad \((-)_{\perp}\), the \(IO\) monad, the error monad, the continuations monad, etc.
Monad Morphisms

A monad morphism from

\[(m_1, \text{fmap}_{m_1}, \text{return}_{m_1}, \text{join}_{m_1})\]

to

\[(m_2, \text{fmap}_{m_2}, \text{return}_{m_2}, \text{join}_{m_2})\]

is a function \(h :: m_1 a \rightarrow m_2 a\) that preserves \text{fmaps}, \text{returns}, and \text{joins}

\[
\begin{align*}
    h \circ \text{fmap}_{m_1} g &= \text{fmap}_{m_2} g \circ h \\
    h \circ \text{return}_{m_1} &= \text{return}_{m_2} \\
    h \circ \text{join}_{m_1} &= \text{join}_{m_2} \circ h \circ \text{fmap}_{m_1} h
\end{align*}
\]
Effectful Lists

- A common generalization of $List_{io}$ and $List_{lazy} \ a$ is

\[
\begin{align*}
\text{data } List' \ m \ a & \quad \text{newtype } List \ m \ a = \\
& = Nil_m \\
& \mid \text{Cons}_m \ a \ (List \ m \ a)
\end{align*}
\]
Effectful Lists

- A common generalization of $List_{io}$ and $List_{lazy} \ a$ is

\[
\text{data } List' \ m \ a \\
= \ \text{Nil}_m \\
| \ \text{Cons}_m \ a \ (List \ m \ a)
\]

- A further generalization replaces list constructors with an arbitrary functor $f$ that describes the data to be interleaved with the effects of the monad $m$:

\[
\text{data } MuFM' \ f \ m \\
= \ \text{In} \ (f \ (MuFM \ f \ m)) \\
| \ \text{Mu} \ (m \ (MuFM' \ f \ m))
\]
Effectful Lists

- A common generalization of \( \text{List}_{io} \) and \( \text{List}_{\text{lazy}} \) \( a \) is

\[
\text{data } \text{List}' m a = \\
\quad \text{newtype } \text{List} m a = \\
\quad \quad \quad \text{Nil}_m \\
\quad \quad \quad \text{List} (m (\text{List}' m a)) \\
\quad \quad | \quad \text{Cons}_m a (\text{List} m a)
\]

- A further generalization replaces list constructors with an arbitrary functor \( f \) that describes the data to be interleaved with the effects of the monad \( m \):

\[
\text{data } \text{MuFM}' f m = \\
\quad \text{newtype } \text{MuFM} f m = \\
\quad \quad \quad \text{In} (f (\text{MuFM} f m)) \\
\quad \quad | \quad \text{Mu} (m (\text{MuFM}' f m))
\]

- \( \text{MuFM} \) represents a pure inductive type described by \( f \) interleaved with effects given by \( m \)
An Append Function for Effectful Lists

- Assume \((\mu(ListF a \circ m), in)\) exists
An Append Function for Effectful Lists

- Assume \( \mu(List\mathcal{F} a \circ m), \text{in} \) exists
- \( List\ m\ a \) is isomorphic to \( m(\mu(List\mathcal{F} a \circ m)) \)
An Append Function for Effectful Lists

- Assume \((\mu(\text{List}F \ a \circ m), \text{in})\) exists
- \(\text{List} \ m \ a\) is isomorphic to \(m(\mu(\text{List}F \ a \circ m))\)
- We can define \(e\text{Append}\) by

\[
\begin{align*}
e\text{Append} & \colon m(\mu(\text{List}F \ a \circ m)) \to m(\mu(\text{List}F \ a \circ m)) \to m(\mu(\text{List}F \ a \circ m)) \\
e\text{Append} \ \text{xs} \ \text{ys} & = \text{join}_m (\text{fmap}_m (|k|) \ \text{xs}) \\
\text{where} \ k \ \text{Nil} & = \ \text{ys} \\
k (\text{Cons} \ a \ \text{xs}) & = \text{return}_m (\text{in} (\text{Cons} \ a (\text{join}_m \ \text{xs})))
\end{align*}
\]
An Append Function for Effectful Lists

- Assume $(\mu(\text{List}F\ a \circ m),\ in)$ exists
- $\text{List} m\ a$ is isomorphic to $m (\mu(\text{List}F\ a \circ m))$
- We can define $e\text{Append}$ by

\[
e\text{Append} :: m (\mu(\text{List}F\ a \circ m)) \rightarrow m (\mu(\text{List}F\ a \circ m)) \rightarrow m (\mu(\text{List}F\ a \circ m))
\]

\[
e\text{Append}\ xs\ ys = join_m (fmap_m (|k|)\ xs)
\]

where

\[
k\ \text{Nil} = ys
\]

\[
k (\text{Cons}\ a\ xs) = return_m (in (\text{Cons}\ a (join_m\ xs)))
\]

- This is similar to the definition of $\text{append}$, but we have had to insert uses of the monadic structure $\text{return}_m$, $\text{join}_m$ and $\text{fmap}_m$ because the initial $f$-algebra is unaware of the presence of effects
Equational Properties of \textit{eAppend}

- Unfolding the definitions gives these \textit{equational properties} of \textit{eAppend}

\begin{align*}
\text{eAppend} \left( \text{return}_m \left( \text{in} \ (\text{Nil}) \right) \right) \ ys &= ys \\
\text{eAppend} \left( \text{return}_m \left( \text{in} \ (\text{Cons} \ a \ xs) \right) \right) \ ys &= \text{return}_m \left( \text{in} \ (\text{Cons} \ a \ (\text{eAppend} \ xs \ ys)) \right)
\end{align*}
Equational Properties of \textit{eAppend}

• Unfolding the definitions gives these \textit{equational properties} of \textit{eAppend}

\[
e\text{Append} \left( \text{return}_m \left( \text{in} \ \text{Nil} \right) \right) \ ys = ys
\]

\[
e\text{Append} \left( \text{return}_m \left( \text{in} \ \text{Cons} \ a \ xs \right) \right) \ ys = \text{return}_m \left( \text{in} \ \text{Cons} \ a \left( e\text{Append} \ xs \ ys \right) \right)
\]

• Deriving these properties takes more work than in the pure case because we have to \textit{shuffle} the \textit{return}_m, \textit{join}_m, and \textit{fmap}_m around in order to apply the monad laws
Equational Properties of $e\text{Append}$

- Unfolding the definitions gives these equational properties of $e\text{Append}$

  $$e\text{Append} \left( \text{return}_m \left( \text{in} \text{ Nil} \right) \right) ys = ys$$

  $$e\text{Append} \left( \text{return}_m \left( \text{in} \left( \text{Cons} \ a \ xs \right) \right) \right) ys = \text{return}_m \left( \text{in} \left( \text{Cons} \ a \left( e\text{Append} \ xs \ ys \right) \right) \right)$$

- Deriving these properties takes more work than in the pure case because we have to shuffle the $\text{return}_m$, $\text{join}_m$, and $\text{fmap}_m$ around in order to apply the monad laws.

- Whenever we use initial $f$-algebras to define functions on data types with interleaved effects, we will repeat this kind of work over again.
Equational Properties of \textit{eAppend}

- Unfolding the definitions gives these \textit{equational properties} of \textit{eAppend}

\[
ed \cdot (\text{return}_m (\text{in} \text{ Nil})) \cdot y = y
\]

\[
ed \cdot (\text{return}_m (\text{in} \text{ Cons} a \cdot x)) \cdot y
\]
\[
= \text{return}_m (\text{in} \text{ Cons} a \cdot (\text{eAppend} \cdot x \cdot y))
\]

- Deriving these properties takes more work than in the pure case because we have to \texttt{shuffle} the \texttt{return}_m, \texttt{join}_m, and \texttt{fmap}_m around in order to apply the monad laws.

- Whenever we use initial \texttt{f}-algebras to define functions on data types with interleaved effects, we will \texttt{repeat} this kind of work over again.

- When we try to prove associativity of \textit{eAppend} we will be \texttt{unable to directly use these properties} as we did in the uneffectful proof because we are forced to unfold the definition of \textit{eAppend} to apply PP1.
Theorem: For all \( xs, ys, zs :: m(\mu(ListF \circ a \circ m)) \),

\[
eAppend xs (eAppend ys zs) = eAppend (eAppend xs ys) zs
\]
**Associativity of eAppend (I)**

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

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

**Proof:**

1. Unfold the definition of $eAppend$ to rewrite LHS to

   $$\text{join}_m (\text{fmap}_m (\lambda k_{eAppend \ ys \ zs}) \ xs)$$

   Here, $k_l$ is the instance of the function $k$ defined in the body of $eAppend$ with the free variable $ys$ replaced by $l$. 
**Theorem:** For all \( xs, ys, zs :: m (\mu(ListF a \circ m)) \),

\[
eAppend xs (eAppend ys zs) = eAppend (eAppend xs ys) zs
\]

**Proof:**

1. Unfold the definition of \( eAppend \) to rewrite LHS to

\[
join_m (fmap_m (k_{eAppend ys zs}) xs)
\]

   Here, \( k_l \) is the instance of the function \( k \) defined in the body of \( eAppend \) with the free variable \( ys \) replaced by \( l \).

2. Use the definition of \( eAppend \) (thrice!), plus naturality of \( join_m \), the third monad law, and the fact that \( fmap_m \) preserves composition to rewrite RHS to

\[
join_m (fmap_m ((\lambda l. eAppend l zs) \circ (k_{ys}) xs)
\]
3. Instantiate Proof Principle 1 and prove the equation

\[(k_{eAppend\;ys\;zs}) = (\lambda l. eAppend\;l\;zs) \circ (k_{ys})\]
3. Instantiate Proof Principle 1 and prove the equation

\[(|k_{eAppend \, ys \, zs}|) = (\lambda l. \, eAppend \, l \, zs) \circ (|k_{ys}|)\]

4. It suffices to prove that for all \(x :: \text{ListF} \, a \, (m \, (\mu (\text{ListF} \, a \, \circ \, m)))\)

\[eAppend \, (|k_{ys}| \, (\text{in} \, x)) \, zs \]

\[= \, k_{eAppend \, xs \, ys} \, (\text{fmap}_{\text{ListF} \, a} \, (\text{fmap}_{m} \, ((\lambda l. \, eAppend \, l \, zs) \circ (|k_{ys}|))) \, x)\]
3. Instantiate Proof Principle 1 and prove the equation

\[(|k_{\text{eAppend}}\ ys\ zs|) = (\lambda l. \text{eAppend } l\ zs) \circ (|k_{ys}|)\]

4. It suffices to prove that for all \(x :: \text{ListF } a\ (m (\mu (\text{ListF } a \circ m)))\)

\[
\text{eAppend } (|k_{ys}| (\text{in } x))\ zs
= k_{eAppend\ xs\ ys} (\text{fmap}_{\text{ListF } a} (\text{fmap}_{m} ((\lambda l. \text{eAppend } l\ zs) \circ (|k_{ys}|)))\ x)
\]

5. Use case analysis according as \(x = \text{Nil}\) or \(x = \text{Cons } a\ xs\)
Associativity of \( e\text{Append} \) (II)

3. Instantiate Proof Principle 1 and prove the equation

\[
(|k_{e\text{Append}} ys zs|) = (\lambda l. e\text{Append} l zs) \circ (|k ys|)
\]

4. It suffices to prove that for all \( x :: \text{ListF} a (m (\mu (\text{ListF} a \circ m))) \)

\[
e\text{Append} (|k ys| (\text{in} x)) zs
= k_{e\text{Append}} xs ys (fmap_{\text{ListF} a} (fmap_m ((\lambda l. e\text{Append} l zs) \circ (|k ys|))) x)
\]

5. Use case analysis according as \( x = \text{Nil} \) or \( x = \text{Cons} a xs \)

6. For each case, use the definitions of \( e\text{Append}, \ fmap_{\text{ListF} a}, \) and the instances of \( k; \) the fact that \( |h| \) is a \( (\text{ListF} a \circ m) \)-algebra homomorphism for all \( h; \) the naturality of \( \text{join}_m; \) the fact that \( fmap_m \) preserves composition; and the third monad law
Associativity of $e\text{Append} (\Pi)$

3. Instantiate Proof Principle 1 and prove the equation

$$(|k_{e\text{Append} \ ys \ zs}|) = (\lambda l. e\text{Append} \ l \ zs) \circ (|k_{ys}|)$$

4. It suffices to prove that for all $x :: \text{ListF} \ a (m (\mu (\text{ListF} \ a \circ m)))$

$$e\text{Append} \ ((|k_{ys}|) \ (\text{in} \ x)) \ zs$$

$$= \ k_{e\text{Append} \ xs \ ys} \ (\text{fmap}_{\text{ListF} \ a} (\text{fmap}_m ((\lambda l. e\text{Append} \ l \ zs) \circ (|k_{ys}|)))) \ x$$

5. Use case analysis according as $x = \text{Nil}$ or $x = \text{Cons} \ a \ xs$

6. For each case, use the definitions of $e\text{Append}$, $\text{fmap}_{\text{ListF} \ a}$, and the instances of $k$; the fact that $(|h|)$ is a $(\text{ListF} \ a \circ m)$-algebra homomorphism for all $h$; the naturality of $\text{join}_m$; the fact that $\text{fmap}_m$ preserves composition; and the third monad law

The proof is upwards of 25 (complicated) lines long!
Problems and Alternatives

- **Problems:**
  1. Requires non-trivial rewriting in order to apply Proof Principle 1
Problems and Alternatives

- Problems:
  1. Requires non-trivial rewriting in order to apply Proof Principle 1
  2. Requires multiple unfoldings of the definition of $eAppend$ to proceed, forcing calculations to be repeated, preventing equational properties from being used, breaking abstraction layers, ...
Problems and Alternatives

• Problems:
  1. Requires non-trivial rewriting in order to apply Proof Principle 1
  2. Requires multiple unfoldings of the definition of $eAppend$ to proceed, forcing calculations to be repeated, preventing equational properties from being used, breaking abstraction layers, ...

• Alternatives:
  1. Use $eAppend \ xs \ ys = extend \ ((|kys|) \ xs)$, where $extend$ is the (argument-flipped) bind operation for $m$ for quicker reduction to Proof Principle 1
Problems and Alternatives

• Problems:
  1. Requires non-trivial rewriting in order to apply Proof Principle 1
  2. Requires multiple unfoldings of the definition of eAppend to proceed, forcing calculations to be repeated, preventing equational properties from being used, breaking abstraction layers, ...

• Alternatives:
  1. Use eAppend xs ys = extend (\(k_{ys}\) xs), where extend is the (argument-flipped) bind operation for m for quicker reduction to Proof Principle 1
  2. Use fold fusion to prove the goal in bullet point 3 to save effort
Problems and Alternatives

- **Problems:**
  1. Requires non-trivial rewriting in order to apply Proof Principle 1
  2. Requires multiple unfoldings of the definition of \textit{eAppend} to proceed, forcing calculations to be repeated, preventing equational properties from being used, breaking abstraction layers, ...

- **Alternatives:**
  1. Use \textit{eAppend} $xs\ ys = extend ((|k|ys)\ xs)$, where \textit{extend} is the (argument-flipped) \textit{bind} operation for $m$ for quicker reduction to Proof Principle 1
  2. Use \textit{fold} fusion to prove the goal in bullet point 3 to save effort

- But we still have to unfold the definition of \textit{eAppend} and reason using the monad laws, and the pure and effectful parts of the proof still aren’t separated. Most importantly, we still **cannot reuse the reasoning from the proof for the pure case!**
Separating Data and Effects

- Use $f$-and-$m$-algebras, i.e., $f$-algebras that are simultaneously $m$-Eilenberg-Moore algebras
Separating Data and Effects

- Use $f$-and-$m$-algebras, i.e., $f$-algebras that are simultaneously $m$-Eilenberg-Moore algebras

- An $m$-Eilenberg-Moore algebra for a type $a$ describes how to properly incorporate the effects of the monad $m$ into values of type $a$
Separating Data and Effects

- Use \( f \)-and-\( m \)-algebras, i.e., \( f \)-algebras that are simultaneously \( m \)-Eilenberg-Moore algebras
- An \( m \)-Eilenberg-Moore algebra for a type \( a \) describes how to properly incorporate the effects of the monad \( m \) into values of type \( a \)
- The \( f \)-algebra part handles the pure parts of the structure
Separating Data and Effects

- Use $f$-and-$m$-algebras, i.e., $f$-algebras that are simultaneously $m$-Eilenberg-Moore algebras.

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

- The $f$-algebra part handles the pure parts of the structure.

- The $m$-Eilenberg-Moore-algebra part handles the effectful parts, accounting for
  - the correct preservation of potential lack of effects (through the preservation of `return`).
  - the potential merging of effects present between layers of the pure datatype (through the preservation of `join`).
**m-Eilenberg-Moore Algebras**

- An *m*-Eilenberg-Moore algebra is a pair

\[(a, l :: m a \rightarrow a)\]

such that \(l\) preserves the *return* and *join* monad structure

\[
\begin{align*}
\begin{array}{c}
\text{a} \\
\downarrow \text{id} \\
\text{a}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{m(a)} \\ \\
\downarrow \text{fmap}_m l \\
\text{m(a)} \\
\downarrow l \\
\text{a}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{a} \\
\downarrow id \\
\text{a}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{m(a)} \\
\downarrow l
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{a} \\
\downarrow id \\
\text{a}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{m(a)} \\
\downarrow l
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{a} \\
\downarrow id \\
\text{a}
\end{array}
\end{align*}
\]
**m-Eilenberg-Moore Algebras**

- An *m*-Eilenberg-Moore algebra is a pair

\[(a, l :: m a \to a)\]

such that \(l\) preserves the return and join monad structure

\[
\begin{array}{c}
\text{a} \xrightarrow{\text{return}_m} m a \\
\downarrow \quad \downarrow l \\
\text{id} \quad \text{a} \\
\text{id} \quad \text{a} \\
\text{id} \quad \text{a} \\
\text{id} \quad \text{a} \\
\text{id} \quad \text{a} \\
\text{id} \quad \text{a} \\
\text{id} \quad \text{a} \\
\end{array}
\]

- An *m*-Eilenberg-Moore algebra homomorphism is an *m*-algebra homomorphism
An \textit{f-and-m-algebra} is a triple

\[(a, k, l)\]

where

\[k :: f a \rightarrow a\]

\[l :: m a \rightarrow a\]

and \(l\) is an \textit{m-Eilenberg-Moore algebra}
**f-and-m-Algebras**

- An *f-and-m-algebra* is a triple

  $$(a, k, l)$$

  where

  $$k :: f a \to a$$

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

  and $l$ is an $m$-Eilenberg-Moore algebra

- An *f-and-m-algebra homomorphism* from $(a, k_a, l_a)$ to $(b, k_b, l_b)$ is a function $h :: a \to b$ that is simultaneously an $f$-algebra homomorphism and an $m$-algebra homomorphism

  $$h \circ k_a = k_b \circ fmap_f h$$

  $$h \circ l_a = l_b \circ fmap_m h$$
Initial $f$-and-$m$-Algebras

- We write $(\mu(f|m), \text{in}_f, \text{in}_m)$ for the initial $f$-and-$m$-algebra
Initial $f$-and-$m$-Algebras

- We write $(\mu(f|m), \text{in}_f, \text{in}_m)$ for the initial $f$-and-$m$-algebra.

- For every $f$-and-$m$-algebra $(a, k, l)$ there is a unique $f$-and-$m$-algebra homomorphism from the initial $f$-and-$m$-algebra $(\mu(f|m), \text{in}_f, \text{in}_m)$ to $(a, k, l)$.
Initial $f$-and-$m$-Algebras

- We write $(\mu(f|m), \text{in}_f, \text{in}_m)$ for the initial $f$-and-$m$-algebra.

- For every $f$-and-$m$-algebra $(a, k, l)$ there is a unique $f$-and-$m$-algebra homomorphism from the initial $f$-and-$m$-algebra $(\mu(f|m), \text{in}_f, \text{in}_m)$ to $(a, k, l)$.

\[
\begin{align*}
\mu(f|m) & \xrightarrow{\text{fmap}_f (|k|l)} f a \\
\text{in}_f & \downarrow \quad k \\
\mu(f|m) (|k|l) & \xrightarrow{a}
\end{align*}
\]
\[
\begin{align*}
\mu(f|m) & \xrightarrow{\text{fmap}_m (|k|l)} m a \\
in_m & \downarrow \quad l \\
\mu(f|m) (|k|l) & \xrightarrow{a}
\end{align*}
\]

- We denote the unique function from $\mu(f|m)$ to $a$ by $(|k|l)$.
• **Proof Principle 2** Let \((a, k, l)\) be an \(f\)-and-\(m\)-algebra and \(g : \mu(f|m) \to a\) be a function. The equation

\[
(k|l) = g
\]

holds iff \(g\) is simultaneously an \(f\)-algebra homomorphism and an \(m\)-algebra homomorphism.
Proof Principle 2 Let \((a, k, l)\) be an \(f\)-and-\(m\)-algebra and \(g : \mu(f|m) \rightarrow a\) be a function. The equation

\[(k|l) = g\]

holds iff \(g\) is simultaneously an \(f\)-algebra homomorphism and an \(m\)-algebra homomorphism, i.e., iff

\[g \circ in_f = k \circ fmap_f g\]

and

\[g \circ in_m = l \circ fmap_m g\]
A Proof Principle for Effectful Data Types

- **Proof Principle 2** Let \((a, k, l)\) be an \(f\)-and-\(m\)-algebra and \(g : \mu(f|m) \to a\) be a function. The equation

\[
(k,l) = g
\]

holds iff \(g\) is simultaneously an \(f\)-algebra homomorphism and an \(m\)-algebra homomorphism, i.e., iff

\[
g \circ \text{in}_f = k \circ \text{fmap}_f g
\]

and

\[
g \circ \text{in}_m = l \circ \text{fmap}_m g
\]

- **Proof Principle 2** cleanly splits the pure and effectful proof obligations!
Representing \( \text{List } m a \)

- Our data type

\[
\begin{align*}
\text{data } \text{List}' m a &=
\quad \text{newtype } \text{List } m a = \\
\quad = \ \text{Nil}_m \\
\quad | \quad \text{Cons}_m a (\text{List } m a)
\end{align*}
\]

can be represented as the carrier \( \mu(\text{ListF } a|m) \) of the initial \((\text{ListF } a)\)-and-\(m\)-algebra
**Representing \( \text{List } m \, a \)**

- Our data type
  
  \[
  \begin{align*}
  \text{data } \text{List'} \, m \, a & \quad \text{newtype } \text{List } m \, a = \\
  & = \text{Nil}_{m} \\
  & \mid \text{Cons}_{m} \, a \, (\text{List } m \, a)
  \end{align*}
  \]

  can be represented as the carrier \( \mu(\text{ListF} \, a|m) \) of the initial \( (\text{ListF} \, a) \)-and-\( m \)-algebra with

  \[
  \begin{align*}
  \text{in}_{\text{ListF} \, a} &: \text{ListF} \, a \, (\text{List } m \, a) \to \text{List } m \, a \\
  \text{in}_{\text{ListF} \, a} \, \text{Nil} & = \text{List} \, (\text{return}_{m} \, \text{Nil}_{m}) \\
  \text{in}_{\text{ListF} \, a} \, (\text{Cons} \, a \, \mathbf{x}s) & = \text{List} \, (\text{return}_{m} \, (\text{Cons}_{m} \, a \, \mathbf{x}s))
  \end{align*}
  \]
Representing \textit{List }\textit{m }\textit{a}

- Our data type

\begin{verbatim}
data List' m a
  = Nil_m
  | Cons_m a (List m a)
\end{verbatim}

can be represented as the carrier \(\mu(\text{List}_F a|m)\) of the initial \((\text{List}_F a)-\) and \(-m\)-algebra with

\begin{verbatim}
in_{\text{List}_F a} :: \text{List}_F a (\text{List} m a) \rightarrow \text{List} m a
in_{\text{List}_F a} \text{Nil} = \text{List} (\text{return}_m \text{Nil}_m)
in_{\text{List}_F a} (\text{Cons} a xs) = \text{List} (\text{return}_m (\text{Cons}_m a xs))
\end{verbatim}

and

\begin{verbatim}
in_m :: m (\text{List} m a) \rightarrow \text{List} m a
in_m ml = \text{List} (do \{\text{List} x ← ml; x\})
\end{verbatim}
Representing \( List \, m \, a \)

- Our data type

\[
\text{data } List' \, m \, a \\
= Nil_m \\
| Cons_m \, a \, (List \, m \, a)
\]

\[
\text{newtype } List \, m \, a = \\
= List \, (m \, (List' \, m \, a)) \\
| Cons_m \, a \, (List \, m \, a)
\]

can be represented as the carrier \( \mu(ListF \, a|\, m) \) of the initial \((ListF \, a)\)-and-\(m\)-algebra with

\[
in_{ListF \, a} :: ListF \, a \, (List \, m \, a) \rightarrow List \, m \, a \\
in_{ListF \, a} \, \text{Nil} = List \, (\text{return}_m \, \text{Nil}_m) \\
in_{ListF \, a} \, (\text{Cons} \, a \, xs) = List \, (\text{return}_m \, (\text{Cons}_m \, a \, xs))
\]

and

\[
in_m :: m \, (List \, m \, a) \rightarrow List \, m \, a \\
in_m \, ml = List \, (do \, \{ \text{List} \, x \leftarrow ml; \, x \})
\]

- If not for the \textbf{List} constructor, \( in_m \) would be \textit{join}
A fold for List \( m \, a \)

The fold for \( \mu \, (\text{ListF} \, a|m) \) is defined as a pair of mutually recursive functions, following the structure of the declaration of List \( m \, a \):

\[
(\mid - | - |) \colon (\text{ListF} \, a \, b \to b) \to (m \, b \to b) \to \text{List} \, m \, a \to b
\]

\[
(\mid k | l \mid) = \text{loop}
\]

where \( \text{loop} \colon \text{List} \, m \, a \to b \)

\[
\text{loop} \,(\text{List} \, x) = l \,(fmap \, m \, \text{loop}' \, x)
\]

\[
\text{loop}' \colon \text{List}' \, m \, a \to b
\]

\[
\text{loop}' \, \text{Nil}_m = k \, \text{Nil}
\]

\[
\text{loop}' \,(\text{Cons}_m \, a \, xs) = k \,(\text{Cons} \, a \,(\text{loop} \, xs))
\]
Representing \textit{eAppend} (Again)

- Assume \((\mu(ListF a|m), \text{in}_{ListF a}, \text{in}_m)\) exists
Representing $eAppend$ (Again)

- Assume $(\mu(ListF \ a|\ m), \ in_{ListF \ a}, \ in_{\ m})$ exists
- We can define $eAppend$ by:

$$eAppend :: \mu(ListF \ a|\ m) \to \mu(ListF \ a|\ m) \to \mu(ListF \ a|\ m)$$

$$eAppend \ xs \ ys = (|k|in_{\ m}|) \ xs$$

where $k \ Nil = ys$

$$k \ (Cons \ a \ xs) = in_{ListF \ a} (Cons \ a \ xs)$$
Representing \textit{eAppend} (Again)

- Assume \((\mu (\text{ListF} \ a | m), \text{in}_{\text{ListF} \ a}, \text{in}_m)\) exists
- We can define \textit{eAppend} by:

\[
\begin{align*}
\text{eAppend} & :: \mu (\text{ListF} \ a | m) \rightarrow \mu (\text{ListF} \ a | m) \\
\text{eAppend} \ xs \ ys & = (|k| \text{in}_m|) \ xs \\
\text{where} \quad k \text{ Nil} & = ys \\
k \ (\text{Cons} \ a \ xs) & = \text{in}_{\text{ListF} \ a} \ (\text{Cons} \ a \ xs)
\end{align*}
\]

- This is identical to the definition of pure \textit{append}, except that
  - \text{in}_m is an additional argument to the \textit{fold}
Representing $eAppend$ (Again)

- Assume $(\mu(\text{ListF} \ a|m), \text{in}_{\text{ListF} \ a}, \text{in}_m)$ exists

- We can define $eAppend$ by:

\[
eAppend :: \mu(\text{ListF} \ a|m) \rightarrow \mu(\text{ListF} \ a|m) \rightarrow \mu(\text{ListF} \ a|m)
\]
\[
eAppend \ xs \ ys = (|k|\text{in}_m|) \ xs
\]
where $k \text{Nil} = ys$
\[
k (\text{Cons} \ a \ xs) = \text{in}_{\text{ListF} \ a} (\text{Cons} \ a \ xs)
\]

- This is identical to the definition of pure $append$, except that
  - $\text{in}_m$ is an additional argument to the $fold$
  - $\text{in}_{\text{ListF} \ a} :: \text{ListF} \ a (\text{List} \ m \ a) \rightarrow \text{List} \ m \ a$ (not $\text{ListF} \ a [a] \rightarrow [a]$)
Representing \textit{eAppend} (Again)

- Assume \((\mu(\text{ListF} a|\text{m}), \text{in}_{\text{ListF} a}, \text{in}_m)\) exists

- We can define \textit{eAppend} by:

\[
\text{eAppend} :: \mu(\text{ListF} a|\text{m}) \rightarrow \mu(\text{ListF} a|\text{m}) \rightarrow \mu(\text{ListF} a|\text{m}) \\
\text{eAppend} \; xs \; ys = (|k|\text{in}_m|) \; xs \\
\text{where} \; k \; \text{Nil} = ys \\
k \; (\text{Cons} \; a \; xs) = \text{in}_{\text{ListF} a} \; (\text{Cons} \; a \; xs)
\]

- This is \textit{identical} to the definition of pure \textit{append}, except that
  - \(\text{in}_m\) is an additional argument to the \textit{fold}
  - \(\text{in}_{\text{ListF} a} :: \text{ListF} a (\text{List} m a) \rightarrow \text{List} m a \) \hspace{1cm} \text{(not} \; \text{ListF} a [a] \rightarrow [a])

- In particular, the pure function \(k\) is — except for types — \textit{identical} to the local function in \textit{append}
Equational Properties of $e\text{Append}$ (Again)

- Unfolding the definitions and using the fact that $(k|\text{in}_m)$ is an $f$-and-$m$-algebra homomorphism gives these equational properties, which are identical — except for types — to the ones for $\text{append}$

$$e\text{Append} \ (\text{in}_{\text{ListF} \ a \ \text{Nil}}) \ ys = ys$$

$$e\text{Append} \ (\text{in}_{\text{ListF} \ a \ (\text{Cons} \ a \ xs)}) \ ys = \text{in}_{\text{ListF} \ a} \ (\text{Cons} \ a \ (e\text{Append} \ xs \ ys))$$
Equational Properties of $e\text{Append}$ (Again)

- Unfolding the definitions and using the fact that $(|k|\text{in}_{m})$ is an $f$-and-$m$-algebra homomorphism gives these equational properties, which are identical — except for types — to the ones for $\text{append}$

$$e\text{Append}(\text{in}_{\text{ListF}a}\text{Nil})\ ys = ys$$
$$e\text{Append}(\text{in}_{\text{ListF}a}(\text{Cons }a\ xs))\ ys = \text{in}_{\text{ListF}a}(\text{Cons }a(e\text{Append }xs\ ys))$$

- Moreover, for any fixed $ys$, $\lambda xs. e\text{Append }xs\ ys$ is an $m$-Eilenberg-Moore homomorphism. So for all $x :: m(\mu(\text{ListF}a|m))$

$$e\text{Append}(\text{in}_{m}x)\ ys = \text{in}_{m}(\text{fmap}_{m}(\lambda xs. e\text{Append }xs\ ys)\ x)$$
Equational Properties of $eAppend$ (Again)

- Unfolding the definitions and using the fact that $(|k|\, in_m)$ is an $f$-and-$m$-algebra homomorphism gives these equational properties, which are identical — except for types — to the ones for $append$

  $$
eAppend (in_{ListF\, a} \Nil) \, ys = ys$$
  $$
eAppend (in_{ListF\, a} (\Cons a \, xs)) \, ys = in_{ListF\, a} (\Cons a (eAppend \, xs \, ys))$$

- Moreover, for any fixed $ys$, $\lambda xs. eAppend \, xs \, ys$ is an $m$-Eilenberg-Moore homomorphism. So for all $x :: m (\mu (ListF\, a|m))$

  $$eAppend (in_m \, x) \, ys = in_m (fmap_m (\lambda xs. eAppend \, xs \, ys) \, x)$$

- Unfolding the definition of $in_m$ we see that $eAppend$ always evaluates the effects placed “before” the first element of its first argument
Theorem: For all $xs, ys, zs :: \mu(ListF a|m)$,

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

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

Proof:

1. Instantiate Proof Principle 2 and prove the equation

$$\langle k|in_m\rangle\;xs = eAppend\;(eAppend\;xs\;ys)\;zs$$
**Theorem:** For all $xs, ys, zs :: \mu(ListF a| m)$,

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

**Proof:**

1. Instantiate Proof Principle 2 and prove the equation

   $$(|k|in_m) \ xs = eAppend \ (eAppend \ xs \ ys) \ zs$$

   i.e.,

   $$(|k|in_m) = g$$

   where

   $$g = \lambda xs. eAppend \ (eAppend \ xs \ ys) \ zs$$

   $$k \ Nil = eAppend \ ys \ zs$$

   $$k \ (\text{Cons} \ a \ xs) = in_{ListF \ a} \ (\text{Cons} \ a \ xs)$$
2. It suffices to prove that for all $x :: \text{ListF} a (\mu(\text{ListF} a | m))$,

$$e\text{Append} (e\text{Append} (\text{in}_{\text{ListF} a} x) ys) zs
= k (\text{fmap}_{\text{ListF} a} (\lambda xs. e\text{Append} (e\text{Append} xs ys) zs) x)$$

and

$$e\text{Append} (e\text{Append} (\text{in}_m x) ys) zs
= \text{in}_m (\text{fmap}_m (\lambda xs. e\text{Append} (e\text{Append} xs ys) zs) x)$$
3. The first is — up to renaming and types — exactly the same as the equation we had to show for pure `append` and is proved using the first two equational properties of `eAppend`
3. The first is — up to renaming and types — exactly the same as the equation we had to show for pure `append` and is proved using the first two equational properties of `eAppend`.

4. The second is proved in just 4 lines using the third equational property of `eAppend` (i.e., that $\lambda xs. eAppend xs ys$ is an $m$-Eilenberg-Moore homomorphism for any fixed $ys$), and the facts that such homomorphisms are closed under composition and that $fmap_m$ preserves composition.
Associativity of \(e\text{Append}\) (Again) (III)

3. The first is — up to renaming and types — exactly the same as the equation we had to show for pure \(\text{append}\) and is proved using the first two equational properties of \(e\text{Append}\).

4. The second is proved in just 4 lines using the third equational property of \(e\text{Append}\) (i.e., that \(\lambda xs. e\text{Append} xs ys\) is an \(m\)-Eilenberg-Moore homomorphism for any fixed \(ys\)), and the facts that such homomorphisms are closed under composition and that \(f\text{map}_m\) preserves composition.

The separation of pure and effectful parts ensures that we can reuse the proof for \(\text{append}\), so only have to establish the side condition for effects.
3. The first is — up to renaming and types — exactly the same as the equation we had to show for pure \textit{append} and is proved using the first two equational properties of \textit{eAppend}.

4. The second is proved in just 4 lines using the third equational property of \textit{eAppend} (i.e., that $\lambda xs. \text{eAppend} \; xs \; ys$ is an $m$-Eilenberg-Moore homomorphism for any fixed $ys$), and the facts that such homomorphisms are closed under composition and that $fmap_m$ preserves composition.

The separation of pure and effectful parts ensures that we can reuse the proof for \textit{append}, so only have to establish the side condition for effects. This proof is simpler, shorter, and more intuitive than the $f$-algebra proof!
Limitations

- Proof Principle 2 fails for proving

\[ e\text{Reverse} \ (e\text{Append} \ xs \ ys) = e\text{Append} \ (e\text{Reverse} \ ys) \ (e\text{Reverse} \ xs) \]

for a suitably defined \( e\text{Reverse} :: \mu(\text{ListF} \ a|m) \rightarrow \mu(\text{ListF} \ a|m) \)
Limitations

• Proof Principle 2 fails for proving

\[ e\text{Reverse} \ (e\text{Append} \ xs \ ys) = e\text{Append} \ (e\text{Reverse} \ ys) \ (e\text{Reverse} \ xs) \]

for a suitably defined \( e\text{Reverse} :: \mu(\text{ListF} \ a|m) \rightarrow \mu(\text{ListF} \ a|m) \)

• Intuitively, the LHS will execute all the effects of \( xs \), then those of \( ys \), while the RHS will execute all the effects of \( ys \), then those of \( xs \)
Limitations

- Proof Principle 2 fails for proving

\[ eRevers (eAppend \, xs \, ys) = eAppend (eReverse \, ys) (eReverse \, xs) \]

for a suitably defined \( eReverse :: \mu(\text{ListF} \, a \, | \, m) \rightarrow \mu(\text{ListF} \, a \, | \, m) \)

- Intuitively, the LHS will execute all the effects of \( xs \), then those of \( ys \), while the RHS will execute all the effects of \( ys \), then those of \( xs \)

- Technically, the problem is that \( \lambda xs. eAppend (eReverse \, ys) (eReverse \, xs) \) is not an \( m \)-Eilenberg-Moore-algebra homomorphism for all \( ys \)
The interleaving of data and non-termination effects can be made explicit using initial $f$-and-$m$-algebras by taking $m$ to be the non-termination monad.
The interleaving of data and non-termination effects can be made explicit using initial $f$-and-$m$-algebras by taking $m$ to be the non-termination monad.

In particular, the type $\text{List}_{\text{lazy}}$ is $\mu(\text{ListF } a|m)$-algebra, where $m$ is the non-termination monad.
We can use the initial \((\text{ListF } a)\)-and-\(\text{IO}\)-algebra \(\text{List}_{\text{io}}\) to give \(h\text{GetContents}\) a type that makes its interleaving of data and effects explicit

\[
h\text{GetContents} :: \text{Handle} \to \text{List}_{\text{io}}
\]
- We can use the initial \((ListF a)\)-and-\(IO\)-algebra \(List_{io}\) to give \(hGetContents\) a type that makes its interleaving of data and effects explicit

\[
hGetContents :: Handle \rightarrow List_{io}
\]

- We can implement \(hGetContents\) using Haskell’s standard primitives for performing \(IO\) on handles

\[
hGetContents h = List_{io} \left(\begin{array}{l}
\text{do } \text{iSEOF } \leftarrow h\text{IsEOF } h \\
\text{if } \text{iSEOF } \text{then return}_{io} \text{Nil}_{io} \\
\text{else do } c \leftarrow h\text{GetChar } h \\
\quad \text{return}_{io} (\text{Cons}_{io} c (hGetContents h))
\end{array}\right)
\]
\textbf{f-and-m-Algebras for Interleaved IO Effects}

- We can use the initial \((ListF a)\)-and-\(IO\)-algebra \(List_{\text{io}}\) to give \(hGetContents\) a type that makes its interleaving of data and effects explicit

\[
hGetContents :: \text{Handle} \rightarrow List_{\text{io}}
\]

- We can implement \(hGetContents\) using Haskell’s standard primitives for performing \(IO\) on handles

\[
hGetContents\ h = List_{\text{io}} (\text{do } is\text{EOF} \leftarrow h\text{IsEOF } h \\
\quad \text{if } is\text{EOF} \text{ then return}_{\text{io}} \text{Nil}_{\text{io}} \\
\quad \text{else do } c \leftarrow h\text{GetChar } h \\
\quad \quad \text{return}_{\text{io}} (\text{Cons}_{\text{io}} c (hGetContents\ h)))
\]

- Now \(IO\) errors are reported within the scope of \(IO\) actions, and we have access to the \(IO\) monad to explicitly close the file
Iteratees

Iteratees interleave reading from some input with effects from some monad, eventually yielding some output

\[
\text{data } \text{Reader'} \ m \ a \ b \\
\quad = \ \text{Input} \ (\text{Maybe } a \rightarrow \text{Reader} \ m \ a \ b) \\
\quad \mid \ \text{Yield } b
\]

\[
\text{newtype } \text{Reader} \ m \ a \ b = \\
\quad \text{Reader} \ (m \ (\text{Reader'} \ m \ a \ b))
\]
Iteratees

- **Iteratees** interleave reading from some input with effects from some monad, eventually yielding some output

\[
\text{data } \text{Reader} \, m \, a \, b \\
= \text{Input} \, (\text{Maybe} \, a \rightarrow \text{Reader} \, m \, a \, b) \quad \text{Reader} \, (m \, (\text{Reader}' \, m \, a \, b)) \\
| \quad \text{Yield} \, b
\]

- A value of type \text{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 \( a \)
**Iteratees**

- **Iteratees** interleave reading from some input with effects from some monad, eventually yielding some output

\[
\text{data } \text{Reader'} \ m \ a \ b \\
= \ \text{Input} \ (\text{Maybe} \ a \rightarrow \text{Reader} \ m \ a \ b) \\
\mid \ \text{Yield} \ b
\]

- A value of type \( \text{Reader} \ m \ a \ b \) is some effect described by the monad \( m \), yielding either a result of type \( b \) or a request for input of type \( a \)

- The \( \text{Reader} \ m \ a \ b \) type is the initial \( f \)-and-\( m \)-algebra, where \( f \) is

\[
\text{data } \text{ReaderF} \ m \ a \ b \ x \\
= \ \text{Input} \ (\text{Maybe} \ a \rightarrow x) \\
\mid \ \text{Yield} \ b
\]
Iteratees

- **Iteratees** interleave reading from some input with effects from some monad, eventually yielding some output.

  \[
  \text{data Reader'} \ m \ a \ b = \\
  \quad = \ \text{Input} \ (\text{Maybe} \ a \ → \ \text{Reader} \ m \ a \ b) \quad \text{Reader} \ (m \ (\text{Reader'} \ m \ a \ b)) \\
  \quad | \quad \text{Yield} \ b
  \]

- A value of type \text{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 \( a \).

- The \text{Reader} \ m \ a \ b \ type is the initial \( f \)-and-\( m \)-algebra, where \( f \) is

  \[
  \text{data ReaderF} \ m \ a \ b \ x = \\
  \quad = \ \text{Input} \ (\text{Maybe} \ a \ → \ x) \\
  \quad | \quad \text{Yield} \ b
  \]

- We can use Proof Principle 2 to reason about programs involving iteratees, e.g., to prove that \text{Reader} \ m \ a \ b \ is a monad whenever \( m \) is
Pipes

- The central definition of the pipes library is

\[
\text{data } Proxy \ a' \ a \ b' \ b \ m \ r \\
= \ \text{Request } a' \ (a \to Proxy \ a' \ a' \ b \ b \ m \ r) \\
| \ \text{Respond } b \ (b' \to Proxy \ a' \ a' \ b \ m \ r) \\
| \ M \ m \ (Proxy \ a' \ a \ b' \ b \ m \ r)) \\
| \ \text{Pure } r
\]
Pipes

- The central definition of the pipes library is

```plaintext
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
```

- 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 `m`, and yielding values of type `r`
Pipes

- The central definition of the pipes library is

\[
data Proxy a' a b' b m r \\
= \text{Request } a' (a \rightarrow Proxy a' a b' b m r) \\
| \text{Respond } b (b' \rightarrow Proxy a' a b' b m r) \\
| M m (Proxy a' a b' b m r)) \\
| \text{Pure } r
\]

- 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 \( m \), and yielding values of type \( r \)

- So the \( Proxy \) type adds the possibility of bidirectional requests and responses to the \( Reader \) type
Pipes

- The central definition of the pipes library is

```haskell
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
```

- 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 `m`, and yielding values of type `r`.

- So the `Proxy` type adds the possibility of bidirectional requests and responses to the `Reader` type.

- `Proxy` types are another instance of data interleaved with effects so we can use Proof Principle 2 to reason about programs involving them.
Conclusions

- $f$-algebras are at the wrong level of abstraction for reasoning about data interleaved with effects
Conclusions

- *f*-algebras are at the **wrong level of abstraction** for reasoning about data interleaved with effects

- Filinski and Støvring’s *f*-and-*m*-algebras **generalize** to categories other than CPO
Conclusions

• $f$-algebras are at the wrong level of abstraction for reasoning about data interleaved with effects

• Filinski and Støvring’s $f$-and-$m$-algebras generalize to categories other than CPO

• Initial $f$-and-$m$-algebras are the effectful analogue of initial $f$-algebras
Conclusions

- *f*-algebras are at the **wrong level of abstraction** for reasoning about data interleaved with effects
- Filinski and Støvring’s *f*-and-*m*-algebras **generalize** to categories other than CPO
- Initial *f*-and-*m*-algebras are the **effectful analogue** of initial *f*-algebras
- Initial *f*-and-*m*-algebras **separate pure and effectful concerns**, and thus let us transfer definitional and proof principles from pure to effectful settings and capture implicit interleaving of effects with data in types
Conclusions

• $f$-algebras are at the wrong level of abstraction for reasoning about data interleaved with effects

• Filinski and Støvring’s $f$-and-$m$-algebras generalize to categories other than CPO

• Initial $f$-and-$m$-algebras are the effectful analogue of initial $f$-algebras

• Initial $f$-and-$m$-algebras separate pure and effectful concerns, and thus let us transfer definitional and proof principles from pure to effectful settings and capture implicit interleaving of effects with data in types

• Other effectful data types (iteratees, pipes, etc.) can also be expressed as initial $f$-and-$m$-algebras, making PP2 available for them
Thank You!
Example — An Eilenberg-Moore Algebra for Errors

- An `ErrorM`-Eilenberg-Moore-algebra with carrier `IO a` is given by

  \[
  l :: ErrorM (IO a) \rightarrow IO a
  \]

  \[
  l \ (\text{Ok } \text{ioa}) \ = \ \text{ioa}
  \]

  \[
  l \ (\text{Error } \text{msg}) \ = \ \text{throw } (\text{ErrorCall } \text{msg})
  \]

- The algebra \( l \) propagates normal `IO` actions, and interprets errors using the exception throwing facilities of the Haskell `IO` monad

- The function `throw` and the constructor `ErrorCall` are part of the standard `Control.Exception` module
From Initial \((f \circ m)\)-Algebras to Initial \(f\)-and-\(m\)-Algebras

**Theorem:** 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.

The proof of this theorem gives us a way to implement \(f\)-and-\(m\)-algebras.