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

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
- 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 possiblity 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 possiblity 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 possiblity 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 possiblity 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

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

  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

- 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

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

  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

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

Inductive Data Types with Effects

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

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

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

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

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-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

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**

$$(f, fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b)$$

Here, $fmap$ is assumed to **preserve identities and composition**

Initial Algebras for Pure Data Types (I)

- **Model the individual "layers" of a data type using a functor**

$$(f, \mathit{fmap} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b)$$

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**

$$(a, k :: f\ a \rightarrow a)$$

Initial Algebras for Pure Data Types (I)

- Model the individual "layers" of a data type using a functor

$$(f, fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b)$$

  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

$$(a, k :: f\ a \rightarrow a)$$

- Characterize the data type as the carrier μf of the initial f-algebra

$$(\mu f, in : f(\mu f) \rightarrow \mu f)$$

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 $h :: a \longrightarrow b$ such that

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

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 $h :: a \longrightarrow b$ such that

$$\begin{tikzcd} f\ a \arrow[r, "fmap\ h"] \arrow[d, "k_a"'] & f\ b \arrow[d, "k_b"] \\ a \arrow[r, "h"] & b \end{tikzcd}$$

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

$$\begin{tikzcd} f\,(\mu f) \arrow[r, "fmap\ (\!|k|\!)"] \arrow[d, "in"'] & f\ a \arrow[d, "k"] \\ \mu f \arrow[r, "(\!|k|\!)"] & a \end{tikzcd}$$

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 $h :: a \longrightarrow b$ such that

$$\begin{array}{ccc} f\ a & \xrightarrow{\mathit{fmap}\ h} & f\ b \\ \downarrow{\scriptstyle k_a} & & \downarrow{\scriptstyle k_b} \\ a & \xrightarrow{h} & b \end{array}$$

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

$$\begin{array}{ccc} f(\mu f) & \xrightarrow{\mathit{fmap}\ (\!|k|\!)} & f\ a \\ \downarrow{\scriptstyle \mathit{in}} & & \downarrow{\scriptstyle k} \\ \mu f & \xrightarrow{(\!|k|\!)} & a \end{array}$$

- We denote the unique function from μf to a by $(\!|k|\!)$

Example I — Initial Algebras for Lists

- The functor $ListF\ a$ describes the individual "layers" of a list

$$
\begin{aligned}
&\textsf{data}\ ListF\ a\ x \\
&\quad = \textsf{Nil} \\
&\quad \mid\ \textsf{Cons}\ a\ x
\end{aligned}
\qquad
\begin{aligned}
&fmap :: (x \to y) \to ListF\ a\ x \to ListF\ a\ y \\
&fmap\ g\ \textsf{Nil} &&= \textsf{Nil} \\
&fmap\ g\ (\textsf{Cons}\ a\ xs) &&= \textsf{Cons}\ a\ (g\ xs)
\end{aligned}
$$

Example I — Initial Algebras for Lists

- The functor $\mathit{ListF}\ a$ describes the individual "layers" of a list

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

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

$$
\begin{aligned}
&\mathit{in} :: \mathit{ListF}\ a\ [a] \to [a] \\
&\mathit{in}\ \mathsf{Nil} = [\,] \\
&\mathit{in}\ (\mathsf{Cons}\ a\ \mathit{xs}) = a : \mathit{xs}
\end{aligned}
$$

Example I — Initial Algebras for Lists

- The functor $\mathit{ListF}\ a$ describes the individual "layers" of a list

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

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

$$
\begin{aligned}
&\mathit{in} :: \mathit{ListF}\ a\ [a] \rightarrow [a] \\
&\mathit{in}\ \mathsf{Nil} \qquad\qquad\ = [\,] \\
&\mathit{in}\ (\mathsf{Cons}\ a\ xs) = a : xs
\end{aligned}
$$

- The *fold* for $[a]$ is

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

Example II — Initial Algebras Generically

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

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

Example II — Initial Algebras Generically

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

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

- The type $\mathit{Mu}\ f$ is the carrier of the initial f-algebra with

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

Example II — Initial Algebras Generically

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

$$\mathbf{data}\; Mu\; f = \mathsf{In}\; \{unIn :: f\; (Mu\; f)\}$$

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

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

- The *fold* for $Mu\; f$ can be defined as

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

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., β, from weak initiality) rules and extensionality (i.e., η, from uniqueness) rules for *fold*s; *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 *fold*s; *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

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

holds iff g is an f-algebra homomorphism, *i.e.*, iff

$$g \circ in \;=\; k \circ fmap\, g$$

$$\begin{CD} f\,(\mu f) @>{fmap\, g}>> f\, a \\ @V{in}VV @VV{k}V \\ \mu f @>{g}>> a \end{CD}$$

Representing *append*

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

Representing *append*

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

$$
\begin{aligned}
&\mathit{append} :: \mu(\mathit{ListF}\ a) \rightarrow \mu(\mathit{ListF}\ a) \rightarrow \mu(\mathit{ListF}\ a)\\
&\mathit{append}\ \mathit{xs}\ \mathit{ys} = (\!|k|\!)\ \mathit{xs}\\
&\quad \mathit{where}\ \ k\ \mathsf{Nil} \qquad\qquad\quad = \mathit{ys}\\
&\qquad\qquad\quad k\ (\mathsf{Cons}\ a\ \mathit{xs}) = \mathit{in}\ (\mathsf{Cons}\ a\ \mathit{xs})
\end{aligned}
$$

Representing *append*

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

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

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

$$
\begin{aligned}
\mathit{append}\ (\mathit{in}\ \mathsf{Nil})\ \mathit{ys} &= \mathit{ys}\\
\mathit{append}\ (\mathit{in}\ (\mathsf{Cons}\ a\ \mathit{xs}))\ \mathit{ys} &= \mathit{in}\ (\mathsf{Cons}\ a\ (\mathit{append}\ \mathit{xs}\ \mathit{ys}))
\end{aligned}
$$

Associativity of *append* (I)

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

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

Associativity of *append* (I)

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

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

Associativity of *append* (I)

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

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

   i.e.,

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

   where

   $$\begin{array}{lcl} g & = & \lambda xs.\ append\ (append\ xs\ ys)\ zs \\ k\ \mathsf{Nil} & = & append\ ys\ zs \\ k\ (\mathsf{Cons}\ a\ xs) & = & in\ (\mathsf{Cons}\ a\ xs) \end{array}$$

Associativity of *append* (II)

2. It suffices to prove that

$$g \circ in \ = \ k \circ fmap\, g$$

i.e., that for all $x :: ListF\, a\, (\mu(ListF\, a))$,

$$\begin{aligned} & = \ append\,(append\,(in\ x)\ ys)\ zs \\ & = \ k\,(fmap\,(\lambda xs.\ append\,(append\ xs\ ys)\ zs)\ x) \end{aligned}$$

Associativity of *append* (II)

2. It suffices to prove that

$$g \circ in \; = \; k \circ fmap\, g$$

i.e., that for all $x :: ListF\, a\, (\mu(ListF\, a))$,

$$\begin{aligned} & = \; append\, (append\, (in\; x)\; ys)\; zs \\ & = \; k\, (fmap\, (\lambda xs.\, append\, (append\; xs\; ys)\; zs)\; x) \end{aligned}$$

3. Use case analysis according as $x = \textsf{Nil}$ or $x = \textsf{Cons}\; a\; xs$

Associativity of *append* (II)

2. It suffices to prove that

$$g \circ in \; = \; k \circ fmap\, g$$

   i.e., that for all $x :: ListF\, a\, (\mu(ListF\, a))$,

$$\begin{aligned} & = \; append\,(append\,(in\; x)\; ys)\; zs \\ & = \; k\,(fmap\,(\lambda xs.\, append\,(append\; xs\; ys)\; zs)\; x) \end{aligned}$$

3. Use case analysis according as $x = \textsf{Nil}$ or $x = \textsf{Cons}\, a\, xs$
4. For each case, we directly use the equational properties of *append* and the definitions of g and $fmap$ for $ListF\, a$

Associativity of *append* (II)

2. It suffices to prove that

$$g \circ in \;=\; k \circ fmap\, g$$

   i.e., that for all $x :: ListF\, a\, (\mu(ListF\, a))$,

$$\begin{aligned} &= \; append\,(append\,(in\; x)\; ys)\; zs \\ &= \; k\,(fmap\,(\lambda xs.\, append\,(append\; xs\; ys)\; zs)\; x) \end{aligned}$$

3. Use case analysis according as $x = \textsf{Nil}$ or $x = \textsf{Cons}\, a\, xs$

4. For each case, we directly use the equational properties of *append* and the definitions of g and *fmap* for $ListF\, a$

The proof is straightforward, easy, and short (9 lines)

Monads for Effects

- Model an effect using a monad

$$(m, \mathit{fmap}_m, \mathit{return}_m, \mathit{join}_m)$$

where

$$\begin{aligned}
\mathit{fmap}_m &:: (a \rightarrow b) \rightarrow m\,a \rightarrow m\,b \\
\mathit{return}_m &:: a \rightarrow m\,a \\
\mathit{join}_m &:: m\,(m\,a) \rightarrow m\,a
\end{aligned}$$

Monads for Effects

- Model an effect using a monad

$$(m, fmap_m, return_m, join_m)$$

  where

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

- The monad laws must be satisfied

Monads for Effects

- Model an effect using a monad

$$(m, fmap_m, return_m, join_m)$$

  where

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

- 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

$$(m, \mathit{fmap}_m, \mathit{return}_m, \mathit{join}_m)$$

  where

$$\begin{aligned} \mathit{fmap}_m &:: (a \to b) \to m\,a \to m\,b \\ \mathit{return}_m &:: a \to m\,a \\ \mathit{join}_m &:: m\,(m\,a) \to m\,a \end{aligned}$$

- 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

$$(m_1, fmap_{m_1}, return_{m_1}, join_{m_1})$$

to

$$(m_2, fmap_{m_2}, return_{m_2}, join_{m_2})$$

is a function $h :: m_1\, a \rightarrow m_2\, a$ that preserves *fmaps*, *returns*, and *joins*

$$\begin{aligned} h \circ fmap_{m_1}\, g &= fmap_{m_2}\, g \circ h \\ h \circ return_{m_1} &= return_{m_2} \\ h \circ join_{m_1} &= join_{m_2} \circ h \circ fmap_{m_1}\, h \end{aligned}$$

Effectful Lists

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

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

Effectful Lists

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

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

- 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:

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

Effectful Lists

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

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

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

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

- $MuFM$ 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 $(\mu(ListF\ a \circ m), in)$ exists
- $List\ m\ a$ is isomorphic to $m\ (\mu(ListF\ a \circ m))$

An Append Function for Effectful Lists

- Assume $(\mu(ListF\ a \circ m), in)$ exists
- $List\ m\ a$ is isomorphic to $m\ (\mu(ListF\ a \circ m))$
- We can define $eAppend$ by

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

An Append Function for Effectful Lists

- Assume $(\mu(\mathit{ListF}\ a \circ m), \mathit{in})$ exists
- $\mathit{List}\ m\ a$ is isomorphic to $m\ (\mu(\mathit{ListF}\ a \circ m))$
- We can define $\mathit{eAppend}$ by

$$
\begin{array}{l}
\mathit{eAppend} :: m\ (\mu(\mathit{ListF}\ a \circ m)) \to m\ (\mu(\mathit{ListF}\ a \circ m)) \to m\ (\mu(\mathit{ListF}\ a \circ m)) \\
\mathit{eAppend}\ \mathit{xs}\ \mathit{ys} = \mathit{join}_m\ (\mathit{fmap}_m\ (\!| k |\!)\ \mathit{xs}) \\
\quad \begin{array}{lll}
\mathit{where}\ k\ \mathsf{Nil} & = & \mathit{ys} \\
\qquad\quad\ k\ (\mathsf{Cons}\ a\ \mathit{xs}) & = & \mathit{return}_m\ (\mathit{in}\ (\mathsf{Cons}\ a\ (\mathit{join}_m\ \mathit{xs})))
\end{array}
\end{array}
$$

- This is similar to the definition of append, but we have had to insert uses of the monadic structure return_m, join_m and fmap_m because the initial f-algebra is unaware of the presence of effects

Equational Properties of $eAppend$

- **Unfolding the definitions gives these equational properties of $eAppend$**

$$eAppend\,(return_m\,(in\;\mathsf{Nil}))\;ys \;=\; ys$$

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

Equational Properties of $eAppend$

- **Unfolding the definitions gives these equational properties of $eAppend$**

$$eAppend\,(return_m\,(in\;\mathsf{Nil}))\;ys \;=\; ys$$

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

- **Deriving these properties takes more work than in the pure case because we have to shuffle the $return_m$, $join_m$, and $fmap_m$ around in order to apply the monad laws**

Equational Properties of $eAppend$

- Unfolding the definitions gives these equational properties of $eAppend$

$$eAppend\,(return_m\,(in\,\mathsf{Nil}))\;ys \;=\; ys$$

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

- Deriving these properties takes more work than in the pure case because we have to shuffle the $return_m$, $join_m$, and $fmap_m$ 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 $eAppend$

- Unfolding the definitions gives these equational properties of $eAppend$

$$eAppend\,(return_m\,(in\;\mathsf{Nil}))\;ys \;=\; ys$$

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

- Deriving these properties takes more work than in the pure case because we have to shuffle the $return_m$, $join_m$, and $fmap_m$ 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
- When we try to prove associativity of $eAppend$ we will be unable to directly use these properties as we did in the uneffectful proof because we are forced to unfold the definition of $eAppend$ to apply PP1

Associativity of $eAppend$ (I)

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

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

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

$$join_m\; (fmap_m\; (\llparenthesis k_{eAppend\; ys\; zs} \rrparenthesis)\; xs)$$

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

Associativity of *eAppend* (I)

Theorem: For all $xs, ys, zs :: m\,(\mu(\mathit{ListF}\;a \circ m))$,

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

Proof:

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

   $$join_m\;(fmap_m\;(\llparenthesis k_{eAppend\;ys\;zs} \rrparenthesis)\;xs)$$

   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

   $$join_m\;(fmap_m\;((\lambda l.\;eAppend\;l\;zs) \circ \llparenthesis k_{ys} \rrparenthesis)\;xs)$$

Associativity of *eAppend* (II)

3. Instantiate Proof Principle 1 and prove the equation

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

Associativity of *eAppend* (II)

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 $x :: ListF\ a\ (m\ (\mu(ListF\ a \circ m)))$

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

Associativity of *eAppend* (II)

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 $x :: ListF\; a\,(m\,(\mu(ListF\; a \circ m)))$

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

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

Associativity of $eAppend$ (II)

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 $x :: ListF\ a\ (m\ (\mu(ListF\ a \circ m)))$

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

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

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

Associativity of *eAppend* (II)

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 $x :: ListF\; a\; (m\; (\mu(ListF\; a \circ m)))$

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

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

6. For each case, use the definitions of $eAppend$, $fmap_{ListF\;a}$, and the instances of k; the fact that $(\!|h|\!)$ is a $(ListF\; a \circ m)$-algebra homomorphism for all h; the naturality of $join_m$; the fact that $fmap_m$ 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 $eAppend\,xs\,ys = extend\,((\!|k_{ys}|\!)\,xs)$, 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 *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
- But we still have to unfold the definition of *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

$$(a, l :: m\,a \longrightarrow a)$$

  such that l preserves the *return* and *join* monad structure

$$\begin{tikzcd}
a \arrow[r, "return_m"] \arrow[dr, "id"'] & m\,a \arrow[d, "l"] \\
 & a
\end{tikzcd}$$

$$\begin{tikzcd}
m\,(m\,a) \arrow[r, "join_m"] \arrow[d, "fmap_m\,l"'] & m\,a \arrow[d, "l"] \\
m\,a \arrow[r, "i"] & a
\end{tikzcd}$$

m-Eilenberg-Moore Algebras

- An m-Eilenberg-Moore algebra is a pair

$$(a, l :: m\, a \longrightarrow a)$$

such that l preserves the *return* and *join* monad structure

$$\begin{tikzcd}
a \arrow[r, "return_m"] \arrow[dr, "id"'] & m\,a \arrow[d, "l"] \\
 & a
\end{tikzcd}$$

$$\begin{tikzcd}
m\,(m\,a) \arrow[r, "join_m"] \arrow[d, "fmap_m\, l"'] & m\,a \arrow[d, "l"] \\
m\,a \arrow[r, "i"] & a
\end{tikzcd}$$

- An m-Eilenberg-Moore algebra homomorphism is an m-algebra homomorphism

f-and-m-Algebras

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

$$(a, k, l)$$

where

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

and l is an m-Eilenberg-Moore algebra

f-and-m-Algebras

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

$$(a, k, l)$$

  where

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

  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 $h :: a \rightarrow b$ that is simultaneously an f-algebra homomorphism and an m-algebra homomorphism

$$\begin{aligned} h \circ k_a &= k_b \circ \mathit{fmap}_f\, h \\ h \circ l_a &= l_b \circ \mathit{fmap}_m\, h \end{aligned}$$

Initial f-and-m-Algebras

- **We write $(\mu(f|m), in_f, in_m)$ for the initial f-and-m-algebra**

Initial f-and-m-Algebras

- We write $(\mu(f|m), in_f, in_m)$ 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 $(\mu(f|m), in_f, in_m)$ to (a, k, l)

$$\begin{tikzcd}
f(\mu(f|m)) \arrow[r, "fmap_f\ (\!|k|l|\!)"] \arrow[d, "in_f"'] & f\ a \arrow[d, "k"] \\
\mu(f|m) \arrow[r, "(\!|k|l|\!)"] & a
\end{tikzcd}
\qquad
\begin{tikzcd}
m(\mu(f|m)) \arrow[r, "fmap_m\ (\!|k|l|\!)"] \arrow[d, "in_m"'] & m\ a \arrow[d, "l"] \\
\mu(f|m) \arrow[r, "(\!|k|l|\!)"] & a
\end{tikzcd}$$

Initial f-and-m-Algebras

- We write $(\mu(f|m), in_f, in_m)$ 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 $(\mu(f|m), in_f, in_m)$ to (a, k, l)

$$\begin{tikzcd} f(\mu(f|m)) \arrow[r, "fmap_f\ (\!|k|l|\!)"] \arrow[d, "in_f"'] & f\ a \arrow[d, "k"] \\ \mu(f|m) \arrow[r, "(\!|k|l|\!)"] & a \end{tikzcd} \qquad \begin{tikzcd} m(\mu(f|m)) \arrow[r, "fmap_m\ (\!|k|l|\!)"] \arrow[d, "in_m"'] & m\ a \arrow[d, "l"] \\ \mu(f|m) \arrow[r, "(\!|k|l|\!)"] & a \end{tikzcd}$$

- We denote the unique function from $\mu(f|m)$ to a by $(\!|k|l|\!)$

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

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

$$g \circ in_f \;=\; k \circ fmap_f\, g$$

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 $g : \mu(f|m) \rightarrow a$ 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

$$g \circ in_m \;=\; l \circ fmap_m\, g$$

- Proof Principle 2 cleanly splits the pure and effectful proof obligations!

Representing $List\ m\ a$

- Our data type

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

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

Representing $\mathit{List}\ m\ a$

- Our data type

$$\begin{aligned}
&\textsf{data}\ \mathit{List}'\ m\ a \\
&\quad = \ \textsf{Nil}_m \\
&\quad \mid \ \textsf{Cons}_m\ a\ (\mathit{List}\ m\ a)
\end{aligned}
\qquad
\begin{aligned}
&\textsf{newtype}\ \mathit{List}\ m\ a = \\
&\quad \textsf{List}\ (m\ (\mathit{List}'\ m\ a))
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{in}_{\mathit{ListF}\ a} :: \mathit{ListF}\ a\ (\mathit{List}\ m\ a) \rightarrow \mathit{List}\ m\ a \\
&\mathit{in}_{\mathit{ListF}\ a}\ \textsf{Nil} \qquad\qquad\quad = \textsf{List}\ (\mathit{return}_m\ \textsf{Nil}_m) \\
&\mathit{in}_{\mathit{ListF}\ a}\ (\textsf{Cons}\ a\ \mathit{xs}) = \textsf{List}\ (\mathit{return}_m\ (\textsf{Cons}_m\ a\ \mathit{xs}))
\end{aligned}$$

Representing $List\ m\ a$

- Our data type

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

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

$$\begin{array}{ll} in_{ListF\ a} :: ListF\ a\ (List\ m\ a) \to List\ m\ a & \\ in_{ListF\ a}\ \textsf{Nil} & = \textsf{List}\ (return_m\ \textsf{Nil}_m) \\ in_{ListF\ a}\ (\textsf{Cons}\ a\ xs) & = \textsf{List}\ (return_m\ (\textsf{Cons}_m\ a\ xs)) \end{array}$$

and

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

Representing $List\ m\ a$

- Our data type

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

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

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

and

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

- If not for the **List** constructor, in_m would be $join$

A *fold* for $List\ m\ a$

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

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

Representing *eAppend* (Again)

- Assume $(\mu(ListF\ a|m), in_{ListF\ a}, in_m)$ exists

Representing *eAppend* (Again)

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

$$
\begin{array}{l}
\mathit{eAppend} :: \mu(\mathit{ListF}\ a|m) \rightarrow \mu(\mathit{ListF}\ a|m) \rightarrow \mu(\mathit{ListF}\ a|m) \\
\mathit{eAppend}\ \mathit{xs}\ \mathit{ys} = (\!|k|\mathit{in}_m|\!)\ \mathit{xs} \\
\quad \text{where } \begin{array}{lcl} k\ \mathsf{Nil} & = & \mathit{ys} \\ k\ (\mathsf{Cons}\ a\ \mathit{xs}) & = & \mathit{in}_{\mathit{ListF}\ a}\ (\mathsf{Cons}\ a\ \mathit{xs}) \end{array}
\end{array}
$$

Representing *eAppend* (Again)

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

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

- This is identical to the definition of pure *append*, except that
  - in_m is an additional argument to the *fold*

Representing *eAppend* (Again)

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

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

- This is identical to the definition of pure *append*, except that
  - in_m is an additional argument to the *fold*
  - $in_{ListF\ a} :: ListF\ a\ (List\ m\ a) \rightarrow List\ m\ a \quad (\text{not } ListF\ a\ [a] \rightarrow [a])$

Representing *eAppend* (Again)

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

$$
\begin{aligned}
&\mathit{eAppend} :: \mu(\mathit{ListF}\ a|m) \rightarrow \mu(\mathit{ListF}\ a|m) \rightarrow \mu(\mathit{ListF}\ a|m)\\
&\mathit{eAppend}\ \mathit{xs}\ \mathit{ys} = (\!| k | \mathit{in}_m |\!)\ \mathit{xs}\\
&\quad \text{where } k\ \mathsf{Nil} \qquad\qquad = \mathit{ys}\\
&\qquad\qquad k\ (\mathsf{Cons}\ a\ \mathit{xs}) = \mathit{in}_{\mathit{ListF}\ a}\ (\mathsf{Cons}\ a\ \mathit{xs})
\end{aligned}
$$

- This is identical to the definition of pure *append*, except that
  - in_m is an additional argument to the *fold*
  - $\mathit{in}_{\mathit{ListF}\ a} :: \mathit{ListF}\ a\ (\mathit{List}\ m\ a) \rightarrow \mathit{List}\ m\ a \quad (\text{not } \mathit{ListF}\ a\ [a] \rightarrow [a])$
- In particular, the pure function k is — except for types — identical to the local function in *append*

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*

$$
\begin{aligned}
eAppend\,(in_{ListF\ a}\,\mathsf{Nil})\;ys &= ys \\
eAppend\,(in_{ListF\ a}\,(\mathsf{Cons}\,a\;xs))\;ys &= in_{ListF\ a}\,(\mathsf{Cons}\,a\,(eAppend\;xs\;ys))
\end{aligned}
$$

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*

$$
\begin{array}{lcl}
eAppend\,(in_{ListF\ a}\,\mathsf{Nil})\;ys & = & ys \\
eAppend\,(in_{ListF\ a}\,(\mathsf{Cons}\,a\;xs))\;ys & = & in_{ListF\ a}\,(\mathsf{Cons}\,a\,(eAppend\;xs\;ys))
\end{array}
$$

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

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

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*

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

- 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

Associativity of *eAppend* (Again) (I)

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

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

Associativity of *eAppend* (Again) (I)

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

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

Associativity of $eAppend$ (Again) (I)

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

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

Associativity of $eAppend$ (Again) (II)

2. It suffices to prove that for all $x :: ListF\ a\ (\mu(ListF\ a|m))$,

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

and

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

Associativity of *eAppend* (Again) (III)

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*

Associativity of *eAppend* (Again) (III)

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 *eAppend* (Again) (III)

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

The separation of pure and effectful parts ensures that we can reuse the proof for *append*, so only have to establish the side condition for effects

Associativity of *eAppend* (Again) (III)

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

The separation of pure and effectful parts ensures that we can reuse the proof for *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

  $$eReverse\,(eAppend\;xs\;ys) = eAppend\,(eReverse\;ys)\,(eReverse\;xs)$$

  for a suitably defined $eReverse :: \mu(ListF\;a|m) \rightarrow \mu(ListF\;a|m)$

Limitations

- Proof Principle 2 fails for proving

  $$eReverse\,(eAppend\; xs\; ys) = eAppend\,(eReverse\; ys)\,(eReverse\; xs)$$

  for a suitably defined $eReverse :: \mu(ListF\; a|m) \to \mu(ListF\; a|m)$

- 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

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

  for a suitably defined $eReverse :: \mu(ListF\ a|m) \rightarrow \mu(ListF\ a|m)$

- 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

f-and-m-Algebras for Interleaved Non-termination

- **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**

f-and-m-Algebras for Interleaved Non-termination

- 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 List_{lazy} is $\mu(\mathit{ListF}\ a|m)$-algebra, where m is the non-termination monad

f-and-m-Algebras for Interleaved IO Effects

- 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

$$hGetContents :: Handle \rightarrow List_{io}$$

f-and-m-Algebras for Interleaved IO Effects

- 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

$$\begin{aligned}
hGetContents\ h = \mathsf{List}_{io}\,(&\mathbf{do}\ isEOF \leftarrow hIsEOF\ h \\
&\quad \mathbf{if}\ isEOF\ \mathbf{then}\ return_{io}\ \mathsf{Nil}_{io} \\
&\quad \mathbf{else\ do}\ c \leftarrow hGetChar\ h \\
&\qquad\qquad return_{io}\,(\mathsf{Cons}_{io}\ c\,(hGetContents\ h)))
\end{aligned}$$

f-and-m-Algebras for Interleaved IO Effects

- 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

$$\begin{array}{l} hGetContents\ h = \mathsf{List}_{io}\ (\mathbf{do}\ isEOF \leftarrow hIsEOF\ h \\ \qquad\qquad\qquad\qquad\qquad \mathbf{if}\ isEOF\ \mathbf{then}\ return_{io}\ \mathsf{Nil}_{io} \\ \qquad\qquad\qquad\qquad\qquad \mathbf{else\ do}\ c \leftarrow hGetChar\ h \\ \qquad\qquad\qquad\qquad\qquad\qquad return_{io}\ (\mathsf{Cons}_{io}\ c\ (hGetContents\ h))) \end{array}$$

- 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

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

$$\begin{aligned}&\textbf{newtype}\ \mathit{Reader}\ m\ a\ b =\\ &\quad \textbf{Reader}\ (m\ (\mathit{Reader}'\ m\ a\ b))\end{aligned}$$

Iteratees

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

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

- A value of type $\mathit{Reader}\ m\ a\ b$ 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

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

- A value of type $\mathit{Reader}\ m\ a\ b$ is some effect described by the monad m, yielding either a result of type b or a request for input of type a
- The $\mathit{Reader}\ m\ a\ b$ type is the initial f-and-m-algebra, where f is

$$\begin{aligned}&\textsf{data}\ \mathit{ReaderF}\ m\ a\ b\ x\\ &\quad = \textsf{Input}\ (\mathit{Maybe}\ a \longrightarrow x)\\ &\quad \mid\ \textsf{Yield}\ b\end{aligned}$$

Iteratees

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

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

- A value of type $\mathit{Reader}\ m\ a\ b$ is some effect described by the monad m, yielding either a result of type b or a request for input of type a
- The $\mathit{Reader}\ m\ a\ b$ type is the initial f-and-m-algebra, where f is

$$\begin{aligned} &\textsf{data}\ \mathit{ReaderF}\ m\ a\ b\ x \\ &\quad = \textsf{Input}\ (\mathit{Maybe}\ a \to x) \\ &\quad \mid\ \textsf{Yield}\ b \end{aligned}$$

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

Pipes

- The central definition of the `pipes` library is

$$
\begin{aligned}
&\textsf{\textbf{data}}\ \mathit{Proxy}\ a'\ a\ b'\ b\ m\ r \\
&\quad = \textsf{\textbf{Request}}\ a'\ (a \longrightarrow \mathit{Proxy}\ a'\ a\ b'\ b\ m\ r) \\
&\quad \mid \textsf{\textbf{Respond}}\ b\ (b' \longrightarrow \mathit{Proxy}\ a'\ a\ b'\ b\ m\ r) \\
&\quad \mid \textsf{\textbf{M}}\ m\ (\mathit{Proxy}\ a'\ a\ b'\ b\ m\ r)) \\
&\quad \mid \textsf{\textbf{Pure}}\ r
\end{aligned}
$$

Pipes

- The central definition of the `pipes` library is

$$
\begin{aligned}
&\textsf{data}\ \mathit{Proxy}\ a'\ a\ b'\ b\ m\ r \\
&\quad = \ \textsf{Request}\ a'\ (a \longrightarrow \mathit{Proxy}\ a'\ a\ b'\ b\ m\ r) \\
&\quad \mid \ \textsf{Respond}\ b\ (b' \longrightarrow \mathit{Proxy}\ a'\ a\ b'\ b\ m\ r) \\
&\quad \mid \ \textsf{M}\ m\ (\mathit{Proxy}\ a'\ a\ b'\ b\ m\ r)) \\
&\quad \mid \ \textsf{Pure}\ r
\end{aligned}
$$

- A value of type $\mathit{Proxy}\ a'\ a\ b'\ b\ m\ r$ 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

$$
\begin{aligned}
&\textsf{\textbf{data}}\ Proxy\ a'\ a\ b'\ b\ m\ r\\
&\quad = \ \textsf{\textbf{Request}}\ a'\ (a \longrightarrow Proxy\ a'\ a\ b'\ b\ m\ r)\\
&\quad \mid \ \textsf{\textbf{Respond}}\ b\ (b' \longrightarrow Proxy\ a'\ a\ b'\ b\ m\ r)\\
&\quad \mid \ \textsf{\textbf{M}}\ m\ (Proxy\ a'\ a\ b'\ b\ m\ r))\\
&\quad \mid \ \textsf{\textbf{Pure}}\ r
\end{aligned}
$$

- 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

$$\begin{aligned} &\textsf{data}\ \mathit{Proxy}\ a'\ a\ b'\ b\ m\ r \\ &\quad = \textsf{Request}\ a'\ (a \rightarrow \mathit{Proxy}\ a'\ a\ b'\ b\ m\ r) \\ &\quad \mid \textsf{Respond}\ b\ (b' \rightarrow \mathit{Proxy}\ a'\ a\ b'\ b\ m\ r) \\ &\quad \mid \textsf{M}\ m\ (\mathit{Proxy}\ a'\ a\ b'\ b\ m\ r)) \\ &\quad \mid \textsf{Pure}\ r \end{aligned}$$

- A value of type $\mathit{Proxy}\ a'\ a\ b'\ b\ m\ r$ 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

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

- 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

rom Initial $(f \circ m)$-Algebras to Initial f-and-m-Algebras

Theorem: Let (f, fmap_f) be a functor, and $(m, \mathit{fmap}_m, \mathit{return}_m, \mathit{join}_m)$ be a monad. If we have an initial $(f \circ m)$-algebra $(\mu(f \circ m), \mathit{in})$, then $m\,(\mu(f \circ m))$ is the carrier of an initial f-and-m-algebra

The proof of this theorems gives us a way to implement f-and-m-algebras