Bind Induction: Extracting Monadic Programs from Proofs

Hadi Shafei and James Caldwell

Department of Computer Science
University of Wyoming
Laramie, WY 82070

Abstract. Container types can be modeled as instances of the Haskell MonadPlus type class which support a fold operation. In this paper we present subclasses that extend the MonadPlus type class to support a membership operator. The laws for the EMonadPlus type class specify how membership behaves with respect to the monad and monad plus operators. Using EMonads we are able write and prove properties of generic specifications of containers. In the second part of the paper we present an induction rule for monads we call bind induction. The computational content of the new induction rule is the Monad bind operator and the new proof rule is proved to be sound. Using this new proof rule we are able to extract monadic programs from proofs. We present an example that uses the rule to extract a simple monadic program from a proof of a specification. We have used the Coq theorem prover with the Coq Type Class mechanism to formalize the definitions presented here and to prove many properties of the formalization.

Keywords: Functional Programming, Monad, Program Verification, Program Extraction, Curry-Howard Correspondence, Type Classes.

1 Introduction

Monads [1, 2] provide a means of structuring computations and encapsulating impure computational effects within pure functional programs. Monads are implemented in Haskell as a type class; any type supporting the bind and return (sometimes called unit\(^1\)) operators in a way that satisfies the monad laws may be declared an instance of the type class. Monads are used in Haskell to structure impure effectful computations like IO actions, threading state through a computation, implementing non-deterministic computations and probabilistic computations, and so on. It also turns out that many container types (lists, bags and sets) can be made instances of the Monad type class. These monads have additional structure and are the focus of this paper.

\(^1\) We use return and unit interchangeably here, though return is a keyword in Coq and so the name unit is used in the Coq development.
Even though monads have proved the usefulness in Haskell, formal reasoning about monadic programs is still something of a challenge\[^2\] \[^3\], \[^4\]. It has been pointed out by Gibbons and Hinze \[^4\] that the Monad class itself is rather weak as it only specifies “the very basic general purpose plumbing of sequential composition.” It is possible to instantiate a type class with a concrete type and then use the methods for that type to specify interesting computations. We are interested in the genericity offered by reasoning at the level of type classes and so, like Gibbons and Hinze, we use and define subclasses of Monad.

To that end, we first define a subclass of the Monad type class to support a membership predicate, a fundamental operation in collections. The EMonad (for ϵ-Monad) type class extends Monad by adding a membership predicate and specifies laws stipulating how membership interacts with the monad operators return and bind. For specifying programs about containers we need even more structure and so we also present an EMonadPlus type class that extends MonadPlus to include the membership predicate and the additional laws specifying how membership interacts with the MonadPlus operators mzero and mplus. Adding a membership predicate to the mix provides the means to write extensional specifications of monadic programs.

Our notion of membership in a monad is expressed as an evaluation modality in Pitts evaluation logic \[^5\], \[^6\]. In \[^7\] Manes also presents a notion of membership for collection monads. Our version is couched in a style of specification and programming based on Haskell’s type classes.

In the second part of the paper, we present a proof rule for EMonadPlus elimination which we call Bind Induction. The rule provides generic induction principle on instances of Foldable EMonadPlus type classes whose computational content is a bind operator. This rule provides the means to extract monadic programs directly from proofs.

The bind operator (written in infix form in Haskell as \(\text{m} \gg= \text{f}\)) has the following type:

\[
\text{Ma} \rightarrow (a \rightarrow \text{Mb}) \rightarrow \text{Mb}
\]

In our earliest attempts to design a proof rule for bind induction we were unable to relate the input to the second argument of bind to the monad which is the first argument. In a dependent type theory, we prefer the following:

\[
\Pi \text{m} : \text{Ma}. ((\Sigma x : a. x \in \text{m}) \rightarrow \text{Mb}) \rightarrow \text{Mb}
\]

Given a bind of the form \(\text{m} \gg= \text{f}\), this stronger type captures the notion that the inputs to \(\text{f}\) are not just any elements of type \(a\) but that they are “elements” of \(\text{m}\). Our proof rule is motivated by the idea captured in this stronger type. In this way, we can extract monadic programs from proofs of specifications. This is an extension of the Curry-Howard correspondence to the realm of monadic program terms.

\[^2\] This first came to our attention in a talk by Xavier Leroy at the 2008 Dependently Typed Programming Workshop in Nottingham. He said that they expected to reason about Monads in Coq using the monad laws, but that they ended up doing all their reasoning by the inversion tactic.
We have verified many of the results presented here in the Coq theorem prover. Algebraic structures can be expressed and investigated in dependent type theory in a natural way\cite{8,9}. As a result, a proof assistant based on the dependent type theory is the best candidate for implementing our definitions and verifying our proofs. Coq supports classes \cite{10–13} which provide an elegant structuring mechanism for dealing with hierarchies of algebraic structures. Type classes are a lot like algebraic structures and we have found the Coq class mechanism to work well in formalizing Haskell style type classes\footnote{Coq proofs of most of the theorems stated in this paper are available at: http://www.cs.uwyo.edu/~jlc/papers/epsilon-Monad.v and http://www.cs.uwyo.edu/~jlc/papers/decomposition.v}.

2 Monad Type Classes and Subclasses

The Monad type class is implemented in Coq’s library as a record, but we use Coq’s class feature to reimplement them here. The class mechanism provides the means to implement rather complex class hierarchies. We formalize numerous subclasses of the monad type class including: monads with a membership predicate, MonadPlus, Foldable monads and so on.

Here is how Monad is defined as a class in Coq:

```
Class Monad (M : Type -> Type) := {
  bind : forall A B, M A -> (A -> M B) -> M B;
  unit : forall A , A -> M A;
  bind_assoc : forall A B C (m : M A) (f : A -> M B) (g : B -> M C),
  bind (bind m f) g = bind m (fun i => bind (f i) g);
  right_unit : forall A (m : M A), bind m (@unit A) = m;
  left_unit : forall A B (x : A) (f : A -> M B), bind (unit x) f = f x
}.
```

The Monad class is parameterized by the carrier M. The bind and unit operators are introduced by stating their types. Bind and unit must satisfy the three monad laws. The first law, `bind_assoc`, states that bind operator must be associative. The second and third laws (`right_unit` and `left_unit`) state that `unit` must be a left and the right identity for `bind`. For readers not familiar with Coq, we note that the syntax `@unit` allows us to make explicit the implicit type argument to `unit`. An advantage of the Coq class mechanism over the Haskell type class mechanism is that instances of the class carry proofs that they satisfy the associated laws.

We say an inhabitant of an instance of the MonadPlus type class is pure if it is constructed using only the methods defined in the MonadPlus class. For example, although lists are an instance of the MonadPlus type class, terms constructed using the list constructors `cons` and `nil` are not a monadically pure monadic program while a term like `(unit 1)` is.

In the definition of the Monad type class in Haskell, the additional methods `join` and `fmap` are derived from definitions of `bind` and `unit`. These pairs of
methods are interdefinable and the Coq files for this paper [?] include that proof as well as a number of others relating these operators.

2.1 Epsilon Monads

Many monads instances can sensibly be interpreted as containers. Membership predicates are a fundamental operation on containers. For a monad $m$ of type $M X$ and an element $x : X$, it is natural to ask if $x \in m$; i.e., whether $x$ is “in” $m$. If the monad $m$ has been constructed as a purely monadic program, $x$ can only occur in $m$ if it was inserted by a bind or unit operation.

We call a monad that supports a membership predicate an $\epsilon$Monad (for $\epsilon$Monad). Here it the Coq implementation as a subclass of the Monad class:

```coq
Class EMonad (M : Type -> Type) :=
{ emonad_monad : Monad M;
  epsilon : forall X, X -> M X -> Prop;
  epsilon_unit : forall X (x y : X), epsilon x (unit y) <-> x = y;
  epsilon_bind :
    forall X Y (m : M X) (f : X -> M Y) (y : Y),
    epsilon y (m >>= f) <-> exists (x : X), (epsilon x m) /
                      (epsilon y (f x))
}.
```

By the definition, $\epsilon$Monad is a subclass of Monad. As a result, bind and unit, together with their properties, are inherited by $\epsilon$Monad. The new operator provided by this class is the membership relation epsilon. There are two laws relating epsilon to bind and unit inherited from Monad. The epsilon_unit law stipulates that the only element that belongs to (unit $x$) is $x$. The epsilon_bind law links the inputs of f to the outputs of m. Formally, $y \in (m >>= f)$ if and only if there is some $x \in m$ such that $y \in (f x)$.

List, Maybe and Tree are among the most frequently used monads. We have formally proved [14] that all three of them are instances of $\epsilon$Monad.

Using the EMonad class we can specify and verify our first monadic program using the EMonad type class. The canonical example is the all-pairs program. The specification for this program is, if $m$ and $n$ are $\epsilon$Monads of type $MA$ and $MB$ for some types $A$ and $B$, then there exists an $\epsilon$-Monad of type $MA*B$ such that it consists of all pairs like $\langle x,y \rangle$ where $x$ belongs to $m$, and $y$ belongs to $n$:

**Theorem 1 (All Pairs).**

\[
\forall X,Y : Type. \forall M : Type \rightarrow Type. \forall m : (M X). \forall n : (M Y).
\]

\[
\text{EMonad } M \implies \exists (p : M (X \ast Y)) \forall (x : X) \forall(y : Y). (x, y) \in p \iff x \in m \land y \in n.
\]

Our claim is that the program:

\[
\lambda X Y M n. m >>= \lambda x. n >>= \lambda y. \text{unit} \langle x, y \rangle
\]

satisfies this specification. A Coq proof of this claim can be found in [14].
2.2 Monad Plus

Working for a while with \textit{bind} and \textit{unit}, (or equivalently, \textit{fmap}, \textit{join} and \textit{unit}) reveals that these operators have fairly limited expressiveness. For example, it is impossible to define \textit{append} for lists just in terms of \textit{bind} and \textit{unit}.

To reason about containers more structure is needed. The MonadPlus type class as defined in Haskell is a good choice for this. It introduces the \textit{mplus} operator (which is an abstraction of \textit{append} for lists) and \textit{mzero} (or \textit{fail}) which can be thought of as an abstraction of the empty list.

This is the Coq definition of MonadPlus in Coq:

\begin{verbatim}
Class MonadPlus (M:Type→Type) := {
monad_plus_monad := Monad M;
mzero : forall X, M X;
mplus : forall X, M X → M X → M X;
mzero_bind : forall X Y (f : X → M Y), (mzero X)>>= f = mzero Y;
mzero_mplus_left : forall X (m : M X), mplus (mzero X) m = m;
mzero_mplus_right : forall X (m : M X), mplus m (mzero X) = m;
mplus_assoc : forall X (m n p : M X), mplus m (mplus n p) = mplus (mplus m n) p ;
mplus_bind : forall X Y (m n : M X) (f : X → M Y),
((mplus m n) >>= f) = mplus (m >>= f) ( n >>= f)
}.
\end{verbatim}

As defined, MonadPlus is a subclass of Monad, so \textit{bind} and \textit{unit} and the monad laws are inherited. The new operators in the MonadPlus type class are \textit{mzero} and \textit{mplus} (mathematically we write $\Box$ as an infix operator. The MonadPlus laws describe how \textit{mzero} and $\Box$ must play with the monad operators.

Note that there are different ways to define MonadPlus in the literature [4]. By our definition, Lists are instances of MonadPlus but Maybe is not since the natural definition of $\Box$ for Maybe not distribute over bind. Proofs about the MonadPlus class defined here can be found in [14].

Now we extend the MonadPlus type class to support membership with the new class EMonadPlus:

\begin{verbatim}
Class EMonadPlus (M:Type→Type) := {
 e_monad_plus_monad_plus := MonadPlus M;
e_monad_plus_e_monad := EMonad M;
eplus_epsilont : forall X (x : X)(m n : M X),
epsilon x (mplus _ m n) ↔ ((epsilon x m) ∨ (epsilon x n));
mzero_epsilont : forall X (m : M X),
(m = mzero X) ↔ (forall (x:X), not(epsilon x m))
}.
\end{verbatim}

\footnote{To see this, you must first agree that any reasonable specification of \textit{append} says that if $x \in m$ and $y \in n$ then $x \in (m'append'n)$ and $y \in (m'append'n)$. But also, \textit{Maybe} is an instance of the EMonad type class and yet every instance of the class contains at most one element.}
Note that EMonadPlus is a subclass of MonadPlus, which means bind, unit, mzero, mplus, and the laws relating them are inherited. EMonadPlus is also a subclass of the EMonad, so we get epsilon and the laws relating it to bind and unit for free. This is an instance of multiple inheritance, and Coq’s class mechanism deals with it in an elegant way. The only thing that remains is to state how epsilon is related to mplus and mzero. As defined in the mplus_epsilon law, an element belongs to mplus of two inhabitants of an instance of the MonadPlus if and only if it belongs to at least one of them. Finally, we assert that nothing belongs to mzero, and vice versa.

Defining the membership predicate for MonadPlus enables us relate mzero and mplus in a new way:

**Theorem 2 (Mplus Zero, Both Zero).**

\[
\forall (X : Type) \forall (M : Type \rightarrow Type) \forall (m, n : M X) 
EMonadPlus M \implies 
m \sqcup n = mzero \iff (m = mzero \land n = mzero).
\]

A formal proof of this theorem in Coq is provided in [14]. The (\rightarrow) direction of the proof holds without using laws for epsilon. The (\leftarrow) direction of the proof uses the EMonadPlus laws in an essential way.

We will see in the next section that having the mplus operator is essential when we are defining the proof rule for monadic programs, so mplus operator not only enriches the programming language, but also enables us to extract monadic programs from proofs.

Let us show that List is an instance of the EMonadPlus. Since it is already proved that List is an instance of MonadPlus and EMonad type classes, it is enough to prove that epsilon respects mzero and mplus, but these are just well known theorems about lists stating that nothing belongs to the empty list, and some element belongs to the append of two lists if and only if it belongs to at least one of them [14].

Having all the operators of the MonadPlus and a membership predicate is a powerful tool that lets us reason about monadic programs. As an example we will see that every pure term m in EMonadPlus can be reduced to mzero, (unit x) for some x of the right type, or it can be decomposed into the mplus of two monadic terms such that none of them is mzero. The assumption that m is a pure term composed using EMonadPlus operators is essential since the proof is by induction on the structure of m. This is captured in the specification by constraining m to be a pure with respect to the operations of the EMonadPlus type class. By parametricity[15], m can only be constructed using the

**Lemma 1 (Mplus Decomposition).**

\[
\forall X : Type. \forall M : Type \rightarrow Type.
\forall m : (\forall (Y : Type)(N : Type \rightarrow Type), EMonadPlus N \Rightarrow N Y).
EMonadPlus M \implies 
(m X M) = mzero \lor \exists (x : X) (m X M) = \text{unit } x' \lor 
\exists (n, k : M X) n \neq mzero \land k \neq mzero \land (m X M) = n \Box k
\]
To prove this lemma we defined an inductive data type in Coq that represents the syntax of monadic programs. We also defined some reduction rules which are proved for the actual MonadPlus type class formally. By using these reduction rules and doing induction on the structure of the pure monadic programs we proved the decomposition lemma above. This method is adapted from [16].

Here is the definition of this data structure in Coq:

\[
\text{Inductive monad_syn (X : Type) : Type :=}
\]
\[
| \text{ret : X} \rightarrow \text{monad_syn X} |
\]
\[
| \text{mplus : monad_syn X} \rightarrow \text{monad_syn X} |
\]
\[
| \text{mzero : monad_syn X} |
\]
\[
| \text{bind : forall (Y : Type), monad_syn Y} \rightarrow \text{monad_syn X} \rightarrow \text{monad_syn X}.|
\]

There are four constructors for a pure monadic program, namely \textit{mzero}, \textit{unit}, \textit{mplus} and \textit{bind}. It is in the folklore that every inhabitant of every instance of the MonadPlus type class has a representation in this form [17]. Absent a full representation theorem of this form, the result here applies only to those instances of MonadPlus for which we know this to be true.

The following reduction rules are previously verified properties of the EMonadPlus:

\textbf{Lemma 2 (Reduction Rules).}\n
\[\forall(X, Y : Type)\forall(m, n, p : M X)\forall(f : X \rightarrow M Y)\]
\[
mzero \Box m = m \quad (1)
m \Box mzero = m \quad (2)
mzero >> f = mzero \quad (3)
(unit x) >>= f = (f x) \quad (4)
(m \Box n) >>= f = (m >>= f) \Box (n >>= f) \quad (5)
m >>= (\lambda x. mzero) = mzero \quad (6)
(m \Box n) \Box p = m \Box (n \Box p)
\]

All of these rules are justified previously except the last one. But if we consider the relation between the bind operator and the membership predicate, it is easy to see that no element can be a member of \(m >>= (\lambda x. mzero)\), as a result it must be equal to \(mzero\). We also need four judgment rules to prove the decomposition lemma:

\textbf{Lemma 3 (Judgment Rules).}\n
\[\forall(X : Type)\forall(x : X). (\text{unit x}) \neq mzero. \quad (1)\]
\[\forall(X : Type)(m, n : M X). \quad (2)\]
\[m \Box n = mzero \leftrightarrow (m = mzero \wedge n = mzero), \quad (2)\]
\[\forall(X : Type)\forall(m : M X). m = mzero \lor m \neq mzero \quad (3)\]
\[\forall(X, Y : Type)\forall(m : M X)\forall(f : X \rightarrow M Y). \quad (4)\]
\[m \neq mzero \land (m >>= f = mzero) \quad \rightarrow f = \lambda x. mzero.\]
To justify the first rule, note that \( x \) always belongs to \((\text{unit } x)\), but nothing belongs to \( mzero \), so \( mzero \) can not be equal to \((\text{unit } x)\). The second rule is proved previously as a lemma. The third rule just asserts that we can decide whether a monad is \( mzero \) or not, and the fourth rule can be justified by applying the membership rule for the bind operator.

As mentioned before, our proof of the decomposition lemma is by induction on the structure of monadic terms, and by using the rules mentioned above. A formal proof of this lemma in Coq is provided in [14].

### 2.3 Foldable MonadPlus

To introduce and justify bind-induction, which is the proof rule for extracting programs that contain the bind operator, we need to add the last piece of structure to our definition of MonadPlus: the ability of being folded.

Here is how Foldable-Monad is defined in Coq:

```
Class FoldMonad (M : Type \to Type) := {
  foldMonad Monad M;
  fold : forall X Y, (X \to Y \to Y) \to Y \to (M X) \to Y
}.
```

Foldable monads that also support MonadPlus operations can be defined as a subclass of MonadPlus and foldable-Monad type classes:

```
Class FoldMonadPlus (M : Type \to Type) := {
  fMonadPlus fMonad : FoldMonad M;
  fMonadPlus MonadPlus : MonadPlus M;
  foldProp : forall X (m : M X), fold (fun x=fun y=unit x \square y) (mzero X) m = m
}.
```

The only property required for fold operation is stated as fold-prop in the definition above. By using this property the following lemma can be proved. We will use this lemma later to justify our induction rule.

**Lemma 4 (Fold and Mplus).**

\[
\forall (X : Type) \forall (M : Type \to Type) \forall (m, n : M X)
Fold Monad Plus M \implies
fold f mzero (m \square n) =
(fold f mzero m) \square (fold f mzero n)
\]

where \( f = \lambda x. (\text{unit } x) \square y \)

A formal proof of this lemma is provided in [14].
3 Bind Induction

In this section we introduce a proof rule for bind induction on foldable instances of MonadPlus that support membership. The extract of the rule includes the bind operator (\>\>\=). The rule supports programming with proofs by providing a mechanism to extract bind operators from proofs.

The rule will be used to prove sequents having the following shape:

\[ \Gamma, m : Ma, \Gamma' \vdash \Sigma n : Mb.\phi(n, m, \bar{z}) \]

This shape is a common pattern in \(\forall \exists\) specifications. The induction is on the foldable EMonadPlus \(m\) and we are able to extract a program of the form \((m \>\>\= \lambda x.t)\) where \(t\) is determined by the extracts of the proofs of the hypotheses of the rule.

We simplify this criteria by simply supposing that the property is hereditary, i.e. it is preserved under composition by \(\Box\). A hereditary property is in some sense a “local” property of a monad instance. This is enough to prove soundness of the Bind rule and simplifies the construction of the proof term in the goal of the rule.

**Definition 1 (\(\Box\)-composability).** If \(M\) is a carrier for an instance of MonadPlus, and \(\psi : Mb \rightarrow Ma \rightarrow \text{Prop}\) is a property of instances of \(M\), then we define \(\Box\)-composability as follows:

\[ \Box\text{-compatible}(Ma, \psi) \overset{\text{def}}{=} \forall j, k : Ma. \]
\[ \forall \langle w_j, \rho_j \rangle : \Sigma n : Mb.\psi(n, j). \]
\[ \forall \langle w_k, \rho_k \rangle : \Sigma n : Mb.\psi(n, k). \]
\[ \psi(w_j \Box w_k, j \Box k) \]

Consider the computational content of a proof of \(\Box\)-compatible\((Ma, \psi)\). It is a function of the form \((\lambda j k \langle w_j, \rho_j \rangle \langle w_k, \rho_k \rangle. \rho)\) where \(j\) and \(k\) are instances of \(Ma\), \(\rho_j\) proves \(\psi(w_j, j)\) and \(\rho_k\) proves \(\psi(w_k, k)\) and \(\rho\) is a proof term witnessing \(\psi(w_j \Box w_k, j \Box k)\). Thus, a proof of \(\Box\)-composability yields a function for composing the witnesses for proofs for \(j\) and \(k\) together into evidence for \(\psi(w_j \Box w_k, j \Box k)\).

**Proof Rule 3.1 (Bind Induction)**

Let \(\Gamma\) be a well formed context such that the carrier

\[ (M : Type \rightarrow Type) \in \Gamma \]

where \(M\) is a foldable instance of MonadPlus with compatible membership \((\epsilon)\). Also, suppose \((a : Type) \in \Gamma\) and \((b : Type) \in \Gamma\). \(\phi\) is a proposition possibly containing free variables \(n, m\) and other variable included in the context \(\Gamma\). We write \(\bar{z} = [z_1, \cdots, z_n], n \in \mathbb{N}\) to denote the list of such variables. We write \(\phi[n, m, \bar{z}]\) to indicate that \(n, m\) and the variables in \(\bar{z}\) may occur free in \(\phi\).
\[ \Gamma, m : \text{Ma}; \Gamma' \vdash \Box - \text{compatible}(\text{Ma}, \lambda n. m. \phi[n, m, \bar{z}]) \textbf{ext} V \]

\[ \Gamma, \Gamma'[\text{mzero}] \vdash \phi[\text{mzero}, \text{mzero}, \bar{z}] \textbf{ext} \rho_z \]

\[ \Gamma, m : \text{Ma}, x : a, x \in m, \Gamma'[\text{unit}(x)] \]

\[ \vdash \Sigma n : \text{Mb}. \phi[n, \text{unit}(x), \bar{z}] \textbf{ext} \{t_x, \rho_x\} \]

\[ \Gamma, m : \text{Ma}, \Gamma'[m] \]

\[ \vdash \Sigma n : \text{Mb}. \phi[n, m, \bar{z}] \textbf{ext} \{m \gg= \lambda x. t_x, \rho(m)\} \]

The term \(\rho(m)\), the evidence that \(\{m \gg= \lambda x. t_x\}\) is a witness for \(\Sigma n : \text{Mb}. \phi(n, m, \bar{z})\) is given by the following term.

\[
\rho(m) = \pi_\beta(\text{foldr})
\]

\[
\{\lambda(k, m, \rho_k)\{k, \bar{m}, \rho_k\},
\langle k\bar{k}, m\bar{m}, V k k \{\{m, \rho_k\} \{\bar{m}, \rho_k\}\} \}
\]

\[
\{\text{mzero, mzero, \rho}_z\}
\]

\[
\{\text{fmap}(\lambda x. \{\text{unit}(x), t_x, \rho_x\}) m\}
\]

We prove soundness of rule below in Theorem 3.

Before we consider the extract term \(\rho(m)\) in more detail, we discuss the three hypotheses of the rule. The first is the verification of the \(\Box\)-composability of the property \(\lambda n. m. \phi[n, m, \bar{z}]\). We refer to the computational content of a proof of this hypothesis as \(V\). The second hypothesis requires a proof that the predicate holds for \text{mzero}. Note that \(\Gamma'\) may depend on \(m\) so we replace all occurrences of \(m\) by \text{mzero} there and also in the conclusion \(\phi[\text{mzero}, \text{mzero}, \bar{z}]\). The extract of a proof of this hypothesis has the form \(\rho_z\), where \(\rho_z\) is a proof term verifying \(\phi[\text{mzero}, \text{mzero}, \bar{z}]\). We have eliminated the existential from the antecedent of the sequent because, examining the extract of the conclusion of the rule, we note that \(\text{mzero} \gg= f = \text{mzero}\) (for any \(f\)) and so the extract \(\rho_z\) must verify that \(n\) is \text{mzero}. A proof of the third hypothesis of the rule shows that for all \(x : a\) such that \(x \in m\) the property holds when \(m\) is of the form \(\text{unit}(x)\). The evidence for this is the witness \(t_x : \text{Mb}\) and a proof term \(\rho_x\) which is evidence for \(\phi[t_x, \text{unit}(x), \bar{z}]\).

Now, consider the proof term \(\rho(m)\). It is defined by projecting out the third element of a fold (hence the foldable constraint on \(M\)) which composes evidence for each element of \(m\) into evidence for \(m\). To do this we use \(V\), the evidence for the \(\Box\)-composability of \(\phi\).

The instance of \text{foldr} has the following type.

\[ \text{foldr} : (\tau \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow M \tau \rightarrow \tau \]

where \(\tau = \Sigma y : \text{Ma}. \Sigma n : \text{Mb}. \phi[n, y, \bar{z}]\)

Note that the variables \(\bar{z}\) will not be free in the context \(\Gamma'\) so we will not continue to include them.

The result of evaluating the following term gives the structure being folded.

\[ \text{fmap}(\lambda x. \{\text{unit}(x), t_x, \rho_x\}) m \]

The term \(\rho_x\) arises from the proof of the third hypothesis and may contain free occurrences of the variable \(x\). Thus, for each element \(y \in m\), the term
\( \lambda x. \langle \text{unit}(x), t_x, \rho_x \rangle \) \( y \) evaluates to a triple \( \langle \text{unit}(y), t_y, \rho_y \rangle \) where \( \rho_y : \phi[n, \text{unit}(y)] \). Thus, we have the extract of the proof of hypothesis three (that any monad of the form \( \text{unit}(x) \) for arbitrary \( x \in m \) satisfies the property) to construct evidence for every \( y \in m \).

Now consider the fold function itself.

\( \lambda \langle k, m, \rho_k \rangle \langle \bar{k}, \bar{m}, \rho_{\bar{k}} \rangle (\lambda \bar{k} \cdot k \bar{k} \cdot m \bar{m}, V \bar{k} (m, \rho_k) \langle \bar{m}, \rho_{\bar{k}} \rangle) \)

The idea is to fold the evidence for each element in \( m \), using \( V \) to build evidence for the proposition \( \Sigma n : Mb. \phi[n, m] \). The function takes two arguments, each is a triple. The first element of each triple is a substructure of \( m \) and the third element is the evidence that the sigma property holds for that element, and the second element is the witness for that. The first argument is a triple of the form \( \langle \text{unit}(y), t_y, \rho_y \rangle \) and the second is of the form \( \langle \text{unit}(y_1) \square \cdots \square \text{unit}(y_k), t_{\text{unit}(y_1) \square \cdots \square \text{unit}(y_k)}, \rho(\text{unit}(y_1) \square \cdots \square \text{unit}(y_k)) \rangle \).

Given these inputs, the result of evaluation is a term of the following form:

\( \langle \text{unit}(y) \square \text{unit}(y_1) \square \cdots \square \text{unit}(y_k), t_y \square t_{\text{unit}(y_1) \square \cdots \square \text{unit}(y_k)}, \rho(\text{unit}(y) \square \text{unit}(y_1) \square \cdots \square \text{unit}(y_k)) \rangle \)

Finally, the zero element for the fold is the triple \( \langle mzero, mzero, \rho_z \rangle \) where \( \rho_z \) (arising from the proof of hypothesis two) is the evidence for \( \Sigma n : Mb. \phi[n, mzero] \).

**Theorem 3 (Soundness).** \( \text{Bind} \) is sound.

**Proof:** To prove soundness we assume the hypotheses hold and that the extract terms indeed have the types ascribed to them:

\( \langle H1 \rangle \Gamma, m : Ma, \Gamma' \vdash V : \square - \text{compatible}(Ma, \lambda n. \lambda \bar{m}. \phi[n, m, \bar{m}]) \)

\( \langle H2 \rangle \Gamma, \Gamma'[mzero] \vdash \rho_z : \phi[mzero, mzero, \bar{z}] \)

\( \langle H3 \rangle \Gamma, m : Ma, x : a, x \in m, \Gamma'[\text{unit}(x)] \vdash \langle t_x, \rho_x \rangle : \Sigma n : Mb. \phi[n, \text{unit}(x), \bar{z}] \)

We must show that the extract of the conclusion of the rule has the correct type:

\( \Gamma, m : Ma, \Gamma' \vdash \langle (m >> \lambda x.t_x), \rho(m) \rangle : \Sigma n : Mb. \phi[n, m, \bar{z}] \)

We proceed by applying lemma 1 (Mplus Decomposition lemma)

There are three cases:

\( \text{case: } m = mzero \) In this case we must show

\( \Gamma, \Gamma'[mzero] \)

\( \vdash (mzero >> \lambda x.t_x, \rho(mzero)) : \Sigma n : Mb. \phi[mzero >> \lambda x.t_x, mzero, \bar{z}] \)
But we know that, for all \( f, \text{mzero} \gg= f = \text{mzero} \) so, more simply, we must show
\[
\Gamma, \Gamma'[\text{mzero}]
\vdash (\text{mzero}, \rho(\text{mzero})) : \Sigma n : Mb. \phi[\text{mzero}, \text{mzero}, z]
\]
We know \( \text{mzero} : Mb \) and so it remains to show that
\[
\Gamma, \Gamma'[\text{mzero}] \vdash \rho(\text{mzero}) : \phi[\text{mzero}, \text{mzero}, z]
\]
By the rules of \text{foldr} we have the following:
\[
\rho(\text{mzero}) = \pi_3(\text{foldr} \ldots \langle \text{mzero}, \rho z \rangle \text{mzero}) = \pi_3(\text{mzero}, \text{mzero}, \rho z) = \rho z
\]
By (H2) we know \( \rho z : \phi[\text{mzero}, \text{mzero}, z] \) which completes the proof of this case.

\textbf{(case: } \exists x : a. m = \text{unit}(x)\text{)} In this case we must show:
\[
\Gamma, x : a, \Gamma'[\text{unit}(x)]
\vdash \langle \text{unit}(x) \gg= \lambda x. t_x, \rho(\text{unit}(x)) \rangle : \\
\Sigma n : Mb. \phi[(\text{unit}(x) \gg= \lambda x. t_x), \text{unit}(x), z] 
\]
Note that \( \text{unit}(x) \gg= \lambda x. t_x = t_x \) so instead we show
\[
\Gamma, x : a, \Gamma'[\text{unit}(x)]
\vdash \langle t_x, \rho(\text{unit}(x)) \rangle : \Sigma n : Mb. \phi[t_x, \text{unit}(x), z]
\]
By (H3) we know \( t_x : Mb \) and so it remains to show the following:
\[
\Gamma, x : a, \Gamma'[\text{unit}(x)] \vdash \rho(\text{unit}(x)) : \phi[t_x, \text{unit}(x), z]
\]
By the definition of \( \rho \),
\[
\rho(\text{unit}(x)) = \\
\pi_3(\langle \text{unit}(x) \Box \text{mzero}, t_x \Box \text{mzero}, \\
V \text{unit}(x) \text{mzero} \langle t_x, \rho z \rangle \langle \text{mzero}, \rho z \rangle \rangle) \\
\pi_3(\langle \text{unit}(x), t_x, \rho z \rangle) = \rho z
\]
But then, by (H3) this case holds.

\textbf{(case: } \exists(j, k : Ma) m = j \Box k\text{)} In this case, we get to assume:
\[
(H4) \Gamma, j : Ma, \Gamma'[j] 
\vdash \langle j \gg= \lambda x. t_x, \rho(j) \rangle : \Sigma n : Mb. \phi[n, j, z] \\
(H5) \Gamma, k : Ma, \Gamma'[k] 
\vdash \langle k \gg= \lambda x. t_x, \rho(k) \rangle : \Sigma n : Mb. \phi[n, k, z]
\]
We must show:
\[
\Gamma, j, k : Ma, \Gamma'[j \Box k] 
\vdash \langle (j \Box k) \gg= \lambda x. t_x, \rho(j \Box k) \rangle : \Sigma n : Mb. \phi[(n, j \Box k), z]
\]
Consider again the specification for a function computing all pairs.

The specification above produces three subgoals:

\[ \Gamma, j : Ma, \Gamma'[j] \vdash \rho(j) : \phi[j] \Rightarrow \lambda x.t_x, j, \overline{z} \]

\[ \Gamma, k : Ma, \Gamma'[k] \vdash \rho(k) : \phi[k] \Rightarrow \lambda x.t_x, k, \overline{z} \]

By (H1) we know

\[ \Gamma, j, k : Ma, \Gamma'[j\square k] \]

\[ \vdash (V \ j \ k \ \{t_j, \rho(j)\} \ \{t_k, \rho(k)\} : \Sigma n : Mb.\phi(n, j\square k, \overline{z}) \]

By mplus-binding property in the definition of MonadPlus we know \( \Box \) distributes over bind, thus we have the following:

\[ (j \square k) \Rightarrow \lambda x.t_x \Rightarrow (j \Rightarrow \lambda x.t_x) \square (k \Rightarrow \lambda x.t_x) \]

By the induction hypothesis, we know \( \rho(j) \) and \( \rho(k) \) provide evidence for the correctness of the types of \( j \) an \( k \). By (H1) \( (V \ j \ k \ \{t_j, \rho(j)\} \ \{t_k, \rho(k)\} \) is evidence for \( \phi(j \Rightarrow \lambda x.t_x) \square (k \Rightarrow \lambda x.t_x), j\square k, \overline{z} \).

\( \Box \)

4 Extracting Monadic Programs

Consider again the specification for a function computing all pairs.

\[ \forall (X, Y : Type) \forall (M : Type \rightarrow Type) \forall (m : M X) \forall (p : M Y) \]

\[ \exists (n : M (X * Y)) \forall (x : X) \forall (y : Y) (x, y) \in n \leftrightarrow x \in m \land y \in p \]

Assuming that \( M \) is a foldable instance of an EMonadPlus, this specification can be proved using the proof rule given above, such that the extract of the proof is a program containing the bind operator. Applying Bind rule on \( m \) in the specification above produces three subgoals:

\[ (G1) m : MX, p : MY \vdash V : \Box:\text{compatible}(MX, \lambda n.\phi[n, m, \overline{z}]) \]

\[ (G2) p : MY \vdash \phi[mzero, mzero, \overline{z}] \]

\[ (G3) m : MX, x : a, x \in m, p : MY \]

\[ \vdash \Sigma n : M (X * Y). \phi[n, unit(x), \overline{z}] \]

Where \( \phi \) is \( \forall (x : X) \forall (y : Y) (x, y) \in n \leftrightarrow x \in m \land y \in p \).

To prove the first subgoal, by definition of \( \Box\)-composability, it is enough to show:

\[ m : MX, p : MY \vdash \forall (j : MX) \forall (k : MX) \]

\[ \forall (w_j, p_j) : \Sigma (n : M (X * Y)) \forall (x : X) \forall (y : Y) (x, y) \epsilon n \leftrightarrow x \epsilon j \land y \epsilon k \]

\[ \forall (w_k, p_k) : \Sigma (n : M (X * Y)) \forall (x : X) \forall (y : Y) (x, y) \epsilon n \leftrightarrow x \epsilon k \land y \epsilon p \]

\[ \forall (x : X) \forall (y : Y) (x, y) \epsilon w_j \Box w_k \leftrightarrow x \epsilon j \Box k \land y \epsilon p \]

Now by definition of \( \Sigma \) it suffices to show:

\[ m : MX, p : MY \vdash \forall (j : MX) \forall (k : MX) \forall (w_j, w_k : M(X * Y)) \]

\[ \forall (x : X) \forall (y : Y) (x, y) \epsilon w_j \Box w_k \leftrightarrow x \epsilon j \Box k \land y \epsilon p \]
Which obviously holds by definition of □. To prove the second subgoal, we need to show:

\[ p:MY \vdash \forall(x:X)\forall(y:Y) (x,y) \in mzero \leftrightarrow x \in mzero \land y \in mzero. \]

Which holds because nothing belongs to mzero. For the third subgoal it is sufficient to show:

\[ m:MX, x:a, x \in m, p:MY \vdash \Sigma n:M(X + Y)\forall(x':X)\forall(y':Y) (x',y') \in n \leftrightarrow x' \in (unit x) \land y \in p. \]

Now if BInd is applied again for \( p \) this time, there will be three new subgoals to prove:

\[ \begin{align*}
(G1') & m:MX, p:MY \vdash \Box-compat(MX, \lambda m.\lambda n.\phi'[n,m,\overline{z}]) \\
(G2') & \vdash \phi'[mzero, mzero, \overline{z}]) \\
(G3') & m:MX, x:X, x \in m, p:MY, y:Y, y \in p \vdash \\
& \Sigma n:M(X + Y) \phi'[n, unit(x), \overline{z}]
\end{align*} \]

Where \( \phi' \) is \( \forall(x':X)\forall(y':Y) (x,y) \in n \leftrightarrow x' \in (unit x) \land y \in p \). To prove \( G1' \) it suffices to prove:

\[ m:MX, x:X, x \in m, p:MY \vdash \\
\forall(j:MY)\forall(k:MY) \\
\forall(w_j, \rho_j): \Sigma(n: M(X + Y))\forall(x':X)\forall(y':Y) \\
\langle x',y' \rangle \in n \leftrightarrow x' \in (unit x) \land y \in j \\
\forall(w_k, \rho_k): \Sigma(n: M(X + Y))\forall(x':X)\forall(y':Y) \\
\langle x',y' \rangle \in n \leftrightarrow x' \in (return x) \land y \in k \\
\forall(x':X)\forall(y':Y) \\
\langle x',y' \rangle \in w_j \Box w_k \leftrightarrow x' \in (unit x) \land y \in j \Box k.
\]

Which can be proved by proving the following:

\[ m:MX, x:X, x \in m, p:MY \vdash \\
\forall(j:MY)\forall(k:MY)\forall(w_j, w_k: M(X + Y)) \\
\forall(x':X)\forall(y':Y) \langle x',y' \rangle \in w_j \leftrightarrow x' = x \land y' \in j \rightarrow \\
\forall(x':X)\forall(y':Y) \langle x',y' \rangle \in w_k \leftrightarrow x' = x \land y' \in k \rightarrow \\
\forall(x':X)\forall(y':Y) \\
\langle x',y' \rangle \in w_j \Box w_k \leftrightarrow x = x' \land y' \in j \Box k.
\]

Which follows from the definition of □. The proof of \( G2' \) is similar to the proof of \( G2' \) which uses the fact that nothing belongs to mzero. To prove \( G3' \) it is enough to show:

\[ m:MX, x:a, x \in m, p:MY, y:Y, y \in p \vdash \\
\Sigma n:M(X + Y)\forall(x':X)\forall(y':Y) \\
\langle x',y' \rangle \in n \leftrightarrow x' \in (unit x) \land y' \in (unit y).
\]
Since the only element that belongs to \((\text{unit} \; x)\) is \(x\) itself, this is equivalent to:

\[
m: MX, \; x:a, \; x \in m, \; p: MY, \; y:y \in p \vdash \\
\Sigma n: M(X \ast Y) \forall(x': X) \forall(y': Y) \langle x', y' \rangle \in n \leftrightarrow x' = x \land y' = y.
\]

Which can be proved easily by choosing \(n\) to be \((\text{unit} \langle x, y \rangle)\). The \text{Blnd} rule has been used twice in this proof. The extraction for the first time is the term \((m >\Rightarrow \lambda y.\text{unit} \langle x, y \rangle)\), and the whole extract is the term \((m >\Rightarrow \lambda x.\text{Blnd} \langle x, y \rangle)\).

5 Conclusions and Future Work

The work described here extends the Curry-Howard correspondence to include an elimination rule for Foldable instances of MonadPlus that support a membership predicate. The program term extracted from the rule is a bind operator and the rule has a close resemblance to an induction rule. It provides a kind of generic induction principle for MonadPlus type classes. The Foldable attribute is used to justify the composition of individual proofs into a proof guaranteed to be correct for every inhabitant of every instance. We believe this is one of the most interesting aspects of the rule.

Coq supports a program extraction mechanism [18–20]. To be able to apply bind induction in a Coq proof we will need to extend Coq to include this new rule together with the specified extract. In future work we will explore the mechanisms for extending Coq to include the bind induction rule.

As mentioned in the text, we have not proved a representation theorem relating syntactic terms in \text{monad-syn} to inhabitants of instances of MonadPlus. The proof of the soundness theorem uses the Mplus Decomposition lemma which depends on structural induction over the syntactic embedding of MonadPlus terms provided by \text{monad-syn}. Such a result is in the folklore, Sculthorpe et. al. [16] state a similar result and cite [17] whose work has not been formally verified. In future work we intended to formalize and prove this representation theorem.

Another path to extend the work presented here will be to explore bind induction for the alternate form of the MonadPlus type class that supports the left catch law as an alternative to the left distribution law. The State monad, the IO monad and Maybe all satisfy left catch but not left distribution.

References