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 foldable monads that support the MonadPlus operations together with a membership operation. In this paper we present a new typeclass we call ϵ-Monad that supports a membership operator for instances of the MonadPlus typeclass. The laws for the ϵ-Monad typeclass specify how membership behaves with respect to the monad and monad plus operators. Using ϵ-Monads we are able write specifications of properties of generic containers. We also present an induction rule for monads we call *bind induction*. The new proof rule is proved to be sound. The computational content of the new induction rule is a bind operator, using this rule we are able to extract monadic programs from proofs. We present an example that uses the rule to extract a monadic program from a proof of a specification. We have used the Coq theorem prover to formalize the definitions presented here and to prove properties of the formalization. We rely on the Coq Type Class mechanism for our formalization.

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

1 Introduction

Monads [1, 2] are a means of structuring computations but also, many monads are containers. The Monad operations are $\{bind, return\}$ or equivalently, $\{map, join, return\}$. It can be proved that these two sets operators are equivalent in expressive power, so it is a matter of taste and style to choose one over another. Here we use bind and return in our definition of the monad type class, but later prove that join and map can be defined in terms of bind and return and vuice versa.

Monads are widely used in Haskell to structure non-functional but useful operations like performing IO actions, creating states, doing non-deterministic computations, probabilistic computation and so on. Even though monads ahve proved so useful, formal reasoning about monadic programs is still somthing of a challenge. Hutton and Fulger in [3] verified a tree relabeling program that uses applicative functors to represent states using a method called point-free equational reasoning, but the function used for relabeling a tree in their work uses explicit recursion instead of using monadic structures to implement recursion. This was one of our motivations for this work, and we believe that we found a rule for doing recursion based on the structure of a monadic term which we call bind-induction. Gibbons and Hinze introduced a method to do equational style reasoning about monadic computations in [4]. Unlike Hutton and Fulger's method, which seems to be applicable only to the state monad, they applied their methods to many examples including state monads, MonadPlus (which they call nondeterministic monad) and a probabilistic Monad. Their approach is based on "algebraic theory" rather than the interpretation of monads in the functional programming community.

Following the idea of monads as containers, adding a membership predicate to the definition of Monad type class is helpful in a number of ways.

It provides the means to write extensional specifications of monadic programs. Secondly, having a membership predicate enables us to reason about monadic programs in an easy and straightforward way. Using membership we can verify monadic programs, and formally prove that they satisfy their specifications. This is important because two of the most powerful tools in pure functional programming languages are recursion and monads. To prove a recursive program satisfies its specification, an inductive proof can be used. One can think of an analogous of induction for monadic programs, and our claim is by adding a membership predicate to the definition of a monad we can define such a proof method.

The membership predicate is used to define a proof rule for monad elimination that has an extract term containing a bind operator. In our earliest attemps 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. Bind has the following type:

$$M\,a \to (a \to Mb) \to Mb$$

As a dependent type, we prefer the type:

$$\Pi m\!:\!Ma.\ ((\Sigma x : a.\, x \in m) \to Mb) \to Mb$$

Given a bind of the form $m >\!\!>\!= f$, this stronger type captures the notion that the inputs to f are not just any elements of type a but that they are elements of m. But of course this type does not match the standard type. Our proof rule uses the idea captured in this stronger type to strethgen its hypotheses. In this way, we can extract monadic programs from proofs of specifications and so have extended the Curry-Howard correspondence to the realm of monadic program terms.

To verify our results formally we have used the Coq theorem prover. Algebraic structures can be expressed and investigated in dependent type theory in a natural way[5, 6]. 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 [7–10] 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 type class structure.

2 Monad Type Classes and Extensions

Monad type class is implemented in Coq's library as a record, but we use Coq's class feature to implement monads. 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.

It is also worth mentioning that in the definition of Monad in the Coq's library, bind has the type : $\forall(X, Y : \mathit{Type}), (X \to M\ Y) \to M\ X \to M\ Y$, where M is the monad carrier. But in the standard definition of the bind operator, for example in Haskell, bind takes the monad as its first and the function as its second argument. We swapped the arguments of the bind operator in our definition to remain faithful to Haskell's definition.

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
}.
```

On the first line the monad carrier M is passed as an argument to the type class. Note that M has to be of kind $(\mathit{Type} \to \mathit{Type})$. Then bind and return operators are defined by stating their types. Bind and return operator must satisfy three properties in a monad called monad laws. The first law is that bind operator must be associative, which we called bind-assoc. The second and third laws state that return (unit) must act as the left and the right identity for bind, called left-unit and right-unit respectively in our definition above. Note that here monad is defined based on two operators unit (also called return) and bind, but the other two important operators can be defined in terms of bind and return, as follows:

```
Definition join X (m: M (M X)): M X :=
  m >>= id.

Definition map X Y (f : X -> Y) (m : M X) : M Y :=
  m >>= (fun x => unit (f x)).
```

As mentioned before, the set of operators $\{bind, return\}$ has the same expressive power as the set $\{join, map, return\}$. To prove this claim formally, first we need to show that join and map as defined above satisfy the desired properties. The following two theorems state two of these properties:

Theorem 1 (Maping Identity).

$$\forall (A : Type) \forall (M : Type \to Type) \forall (m : M(MA)) \\ Monad\ M \Rightarrow map\ id_A\ m = m.$$

Theorem 2 (Composing Maps).

$$\forall (X, Y, Z : Type) \forall (f : X \to Y) \forall (g : Y \to Z) \\ \forall (M : Type \to Type) \forall (m : M\ X) \\ Monad\ M \Rightarrow map\ g\ (map\ f\ m) = map\ (g \circ f)\ m.$$

The full list of these theorems about map and join and their formal proofs in Coq can be found in [11].

We also need to show that bind can be defined in term of join and map. This may seem an obviouse fact, but in our case it needs a proof, because we defined join and map in terms of bind and return. A formal proof of this in Coq can be found in [11]

2.1 Epsilon Monads

Monads can (often) be considered to be containers and a fundamental operation on containers is a membership test. For a monad m of type MX it is natural to query whether an element x of type X is in m. If the monad m has been constructed by a pure monadic program x can only occur in m if it was inserted by a bind or return operation. We define the following two axioms that must hold for a membership relation with respect to the monad operations.

$$\begin{array}{ll} \epsilon - \mathsf{return} & \forall (X : Type) \forall (x, y : X)\ x \in (\mathsf{return}\, y) \Leftrightarrow x = y \\ \epsilon - \mathsf{bind} & \forall (X, Y : Type) \forall (M : Monad) \forall (m : M\, X) \\ & \quad \forall (f : X \to M\, Y) \forall (y : Y) \\ & \quad\quad y \in (m >\!\!>\!= f) \Leftrightarrow \exists (x : X)\ x \in m \wedge y \in (f\ x) \end{array}$$

We call a monad that supports a membership predicae an ϵ-Monad, and here is how we implemented it in Coq as a subclass of the Monad class:

```
Class E_Monad (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), and (epsilon x m) (epsilon y (f x))
}.
```

Note that by his definition Monad is a super class of the ϵ-Monad. As a result, bind and return operators and their properties are inherited to the ϵ-Monad. The only new operator of the ϵ-Monad is a membership predicate. There are two rules relating the membership predicate to the bind and return operators. Epsilon-unit states that the only element that belongs to $(return\ x)$ is x. Epsilon-bind expresses the desired relation between epsilon and bind.

Join and map can be defined in a similar way for the ϵ-Monad, but now we need to show that the epsilon predicate respects join and map. The expected relations between join, map and the epsilon predicate are described in the following theorems:

Theorem 3 (Join and Epsilon).

$$\begin{aligned}&\forall(X : Type)\forall(M : Type \to Type)\forall(m : M(M\ X))\forall(x : X)\\ &\quad E - Monad\ M \Rightarrow x \in m \leftrightarrow \exists(n : M\ X)\ x \in n\ \wedge n \in m.\end{aligned}$$

Theorem 4 (Map and Epsilon).

$$\begin{aligned}&\forall(X, Y : Type)\forall(M : Type \to Type)\\ &\quad\forall(m : M\ X)\forall(f : X \to Y)\forall(y : Y)\\ &\quad\quad E - Monad\ M \Rightarrow\\ &\quad\quad\quad y \in (map\ f\ m) \leftrightarrow \exists(x : X)\ x \in m \wedge y = f(x).\end{aligned}$$

A formal proof of these theorems can be found in [11].

List, Maybe and Tree are among the most frequently used monads. We proved that all three of them are instances of the ϵ-Monads.

To make List an instance of the ϵ-Monad, we followed the standard definitions of bind and return and membership for lists. It only remains to show that the membership predicate respects the bind operator, as stated in the following theorem:

Theorem 5 (Bind and Epsilon for List).

$$\begin{aligned}&\forall(X, Y : Type)\forall(m : list\ X)\forall(f : X \to list\ Y)\forall(y : Y)\\ &\quad y \in (my - flat - map\ \ m\ \ f) \leftrightarrow\\ &\quad\quad \exists(x : X)\ x \in m \wedge y \in f(x).\end{aligned}$$

This theorem is also proved formally in Coq, and the proof is provided in [11].

It is a recognized fact that if the Tree data type is defined in a way that the values are stored in the nodes, then it will not be an instance of the Monad type class, at least in a natural way. But if the Tree data type is defined such that the vallues are stored on the leaves, then it can be a monad.

Here is how such a tree is defined in Coq:

```
Inductive Tree (A:Set):Set :=
  | Leaf : A -> Tree A
  | Branch : Tree A -> Tree A -> Tree A.
```

Now to make it an instance of Monad, we need to define return and bind operators. Return is simply a function that takes an element, and returns a Leaf containing only that element:

```
Definition tree_unit (A : Set) (a : A) : Tree A := Leaf a.
```

Following the analogy of defining bind as flat–map, we can define the bind operator on trees as follows:

```
Fixpoint tree_bind (A B:Set) (m : Tree A) (f:A->Tree B) : Tree B :=
  match m with
  | Leaf x => f x
  | Branch l r => Branch (tree_bind l f) (tree_bind r f)
  end.
```

Now three theorems are needed to prove that these definitions actually create an instance of the Monad type class:

Theorem 6 (Right Unit for Tree).

$$\begin{array}{l}\forall(X,Y:Type)\forall(x:X)\forall(f:X\to Tree\,Y)\\ \quad tree-bind\ \ (Leaf\ x)\ \ f=f(x).\end{array}$$

Theorem 7 (Left Unit for Tree).

$$\begin{array}{l}\forall(X:Type)\forall(t:Tree\,X)\\ \quad tree-bind\ \ t\ \ (\lambda x.(Leaf\ x))=t.\end{array}$$

Theorem 8 (Bind Associativity for Tree).

$$\begin{array}{l}\forall(X,Y,Z:Type)\forall(t:Tree\,X)\forall(f:X\to Tree\,Y)\\ \quad\forall(g:Y\to Tree\,Z)\\ \qquad tree-bind\ \ (tree-bind\ \ t\ \ f)\ \ g\ \ =\\ \qquad tree-bind\ \ t\ \ (\lambda i.(tree-bind\ \ f(i)\ \ g).\end{array}$$

Here is how the membership predicate is defined for Tree:

```
Fixpoint in_tree (A : Set)(a : A)(t : Tree A) : Prop :=
  match t with
  | Leaf x => (x=a)
  | Branch l r => or (in_tree a l) (in_tree a r)
  end.
```

Now two other theorems are needed to prove that ϵ respects bind and unit.

Theorem 9 (Unit and Epsilon for Tree).

$$\begin{array}{l}\forall (X : Type) \forall (x, y : X)\\ \quad in - tree\ \ y\ \ (Leaf\ x) \leftrightarrow x = y.\end{array}$$

Theorem 10 (Bind and Epsilon for Tree).

$$\begin{array}{l}\forall (X, Y : Type) \forall (t : Tree\ X) \forall (f : X \to Tree\ Y) \forall (y : Y)\\ \quad in - tree\ \ y\ \ (tree - bind\ \ f\ \ t)\ \ \leftrightarrow\\ \quad \exists (x : X)\ (in - tree\ \ x\ \ t) \wedge (in - tree\ \ y\ \ f(x)).\end{array}$$

Formal proofs of these theorems in Coq can be found in [11]. It is easy to see that Maybe is also an instance of the ϵ-Monad.

Now we can verify our first monadic program using the ϵ-Monad type class. Our canonical example in this paper is the all–pairs program. The specification for this program is, if m and n are two ϵ-Monads of type MA and MB for some types A and B, then there exists an ϵ-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 11 (All Pairs).

$$\begin{array}{l}\forall (X, Y : Type) \forall (M : Type \to Type) \forall (m : M\ X)(n : MY)\\ \quad E - Monad\ M \Longrightarrow\\ \quad\quad \exists (p : M\ (X * Y)) \forall (x : X) \forall (y : Y)\\ \quad\quad\quad \langle x, y \rangle \in p \leftrightarrow x \in m \wedge y \in n.\end{array}$$

Our claim is that the program:

$m >\!\!>\!= \lambda x.\, n >\!\!>\!= \lambda y.\, return\, \langle x, y \rangle$

satisfies this specification. We formally proved this claim in Coq, and the proof in provided in [11].

2.2 Monad Plus

Working for a while with bind and return, or equivalently, map, join and return reveals to the programmer that the power of these operators is very restricted. As an example, it is impossible to define append just in terms of bind and return. More formally, if we call the programs that only consist of bind and return (and join and map) pure monadic programs, it is impossible to write a pure monadic program that satisfies the specification of append. Here is an informal argument for this claim:

If append is definable for every ϵ-Monad, then for every ϵ-Monad we can construct the term $(return\ x) + +(return\ y)$, where x and y are two distinct elements and $++$ denotes append. Then it follows from the natural specification of append that for every instance of the ϵ-Monad say M, and every type X with at least two distinct element, there exists a monad of type MX such that it has two distinct elements. On the other hand, by definition of the membership predicate for the Maybe type class, we know that it is an instance of the ϵ-Monad, and each member of it has at most 1 element, which is a contradiction.

As a result, it seems a reasonable approach to extend our investigation to monads that possess an operator similar to append. MonadPlus type class as defined in Haskell is a good choice for this. It has an operator named mplus which is an abstraction of the append for lists, and mzero (or fail) which can be seen as an abstraction of the empty list.

This is how we defined MonadPlus type class in Coq:

```
Class Monad_Plus (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)
}.
```

Notice that MonadPlus is defined as a subclaas of Monad, so bind and return will be inherited to MonadPlus. The new operators in the MonadPlus type class are mzero and *mplus*. Binding mzero with any monadplus must result in mzero as stated in mzero-mplus-left rule. Mzero must be left and right identity for mplus, called mzero-mplus-left and mzero-mplus right in our definition respectively. Another property of MonadPlus is that mplus operator must be associative, which is stated as mplus-assoc in the definition above. Finally, mplus distributes over bind, called mplus-bind.

Note that there are different ways to define MonadPlus in the literature, and by choosing different definition, different monads may or may not be instances of the MonadPlus type class. For example in our definition List is a MonadPlus but Maybe is not.

If we define mplus and mzero to be the append operator and the empty list for the List monad, it is straightforward to check that List is a MonadPlus [11].

It is worth mentioning that Maybe is almost an instance of MonadPlus. If mzero is defined to be *Nothing*, and mplus is defined as follows:

Definition 1 (Mplus for Maybe).
$\forall(X : Type)(x, y : X)(m : Maybe\ X)$

$$(1)\ \ mplus\ \ Nothing\ \ m = mplus\ \ m\ \ Nothing = m.$$

$$(2)\ \ mplus\ \ (Just\ \ x)\ \ (Just\ \ y) = Just\ \ y.$$

Then Maybe satisfies the left and right identity laws, and it also satisfies left–zero law, but mplus does not distribute over the bind.
Tree as defined above can not be an instance of MonadPlus, since there is no way to define *mzero*.

A question that may arise is: what is the relation between *mplus* and *join* and *map* in an arbitrary instance of the MonadPlus? The answer is given in the following two theorems. Note that $\Box$ symbol represents the *mplus* operaor.

Theorem 12 (Mplus Distributes Over Join).

$$\begin{array}{l}\forall(X : Type)\forall(M : Type \to Type)\forall(m, n : M(M\ X))\\ \quad MonadPlus\ M \Rightarrow\ join\ (m\Box n) = (join\ \ m)\Box(join\ \ n).\end{array}$$

Theorem 13 (Mplus Distributes Over Map).

$$\begin{array}{l}\forall(X, Y : Type)\forall(M : Type \to Type)\forall(m, n : M\ X)\\ \ \ \forall(f : x \to M\ X)\\ \quad MonadPlus\ M \Rightarrow\\ \quad\ \ map\ \ f\ \ (m\Box n) = (map\ \ f\ \ m)\Box(map\ \ f\ \ n).\end{array}$$

Formal proofs of these theorems in Coq are provided in [11].

Now we can extend ϵ to the MonadPlus type class to define a type class that supports mzero, mplus and a membership predicate. We call this type class ϵ-MonadPlus:

```
Class E_Monad_Plus (M:Type->Type) :=
{
  e_monad_plus_monad_plus :> Monad_Plus M;
  e_monad_plus_e_monad :> E_Monad M;
  mplus_epsilon:forall X (x : X)(m n : M X),
               epsilon x (mplus _ m  n) <->
               ((epsilon x m) \/ (epsilon x n));
  mzero_epsilon:forall X (m : M X),
                (m = mzero X) <->
                (forall (x:X), not(epsilon x m))
}.
```

Note that ϵ-MonadPlus is a subclass of MonadPlus, which means bind, return, mzero, mplus and their rules are inherited to the ϵ-MonadPlus. It is also stated that ϵ-MonadPlus is a subclass of the ϵ-Monad, so we also get ϵ and its relation with bind and return for free. One can see that here we have multiple inheritence, and using Coq's classes to implement monads allows us to deal with inheritence in a nice and elegant way. The only thing that remains is to state how ϵ is related to mplus and mzero. As defined in mplus-epsilon rule above, an element belongs to mplus of two inhabitants of an instance of the MonadPluses if and only if it belongs to at least one of them. Finally, we asserted that nothing belongs to *mzero*, and vice versa. Note that we are following the analogy of the append and the empty list for these definitions.

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

Theorem 14 (Mplus Zero,Both Zero).

$$\begin{array}{l}\forall(X : Type)\forall(M : Type \to Type)\forall(m, n : M\ X)\\ \quad E{-}MonadPlus\ M \Rightarrow\\ \quad\quad m \,\square\, n = mzero \leftrightarrow (m = mzero \wedge n = mzero).\end{array}$$

A formal proof of this theorem in Coq is provided in [11].

We will see in the next chapter 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 ϵ-MonadPlus. Since it is already proved that List is an instance of MonadPlus and ϵ-Monad type classes, it is enough to prove that ϵ 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 [11].

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 monadic term can be reduced to mzero,(return 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.

Lemma 1 (Mplus Decomposition).

$$\begin{array}{l}\forall(X : Type)\forall(M : Type \to Type)\forall(m : M\ X)\\ \quad E - MonadPlus\ M \Longrightarrow\\ \quad m = mzero \vee \exists(x : X)\ m = return\ x \vee\\ \quad\ \exists(n, k : M\ X)\ n \neq mzero \wedge k \neq mzero \wedge m = n \square k\end{array}$$

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 [12].

Here is the definition of this data structure in Coq:

```
Inductive monad_syn (X : Type) : Type :=
  | ret : X -> monad_syn X
  | mplus : monad_syn X -> monad_syn X
            -> monad_syn X
  | mzero : monad_syn X
  | bind : forall (Y :Type),
           monad_syn Y ->
           (Y -> monad_syn X) ->
            monad_syn X.
```

There are four constructors for a pure monadic program, namely *mzero*, *return*, *mplus* and *bind*. It is in the folklore that every inhabitant of every instance of the MonadPlus type class has a representation in this form [13]. 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 ϵ-MonadPlus:

Lemma 2 (Reduction Rules).
$\forall (X, Y : Type) \forall (m, n, p : M\ X) \forall (f : X \rightarrow M\ Y)$

$$
\begin{array}{lll}
mzero \,\square\, m & \Longrightarrow & m \qquad (1) \\
m \,\square\, mzero & \Longrightarrow & m \qquad (2) \\
mzero >\!\!>\!= f & \Longrightarrow & mzero \qquad (3) \\
(return\, x) >\!\!>\!= f & \Longrightarrow & (f\ \ x) \qquad (4) \\
(m \,\square\, n) >\!\!>\!=\ f & \Longrightarrow & (m >\!\!>\!= f) \square (n >\!\!>\!= f) \qquad (5) \\
m >\!\!>\!= (\lambda x.mzero) & \Longrightarrow & mzero \qquad (6) \\
(m \,\square\, n) \,\square\, p & \Longrightarrow & m \,\square\, (n \,\square\, p)
\end{array}
$$

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:

Lemma 3 (Judgment Rules).

$$\begin{aligned}
&\forall(X : Type)\forall(x : X).(return\ x) \neq mzero. && (1)\\
&\forall(X : Type)(m, n : M\ X). && (2)\\
&\quad m \mathbin{\Box} n = mzero \leftrightarrow (m = mzero \wedge n = mzero). &&\\
&\forall(X : Type)\forall(m : M\ X).m = mzero \vee m \neq mzero && (3)\\
&\forall(X, Y : Type)\forall(m : M\ X)\forall(f : X \to M\ Y). && (4)\\
&\quad (m \neq mzero \wedge (m >\!\!>\!= f = mzero)) &&\\
&\quad \to f = \lambda x.mzero. &&
\end{aligned}$$

To justify the first rule, note that x always belongs to $(return\ x)$, but nothing belongs to $mzero$, so $mzero$ can not be equal to $(return\ x)$. 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 [11].

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 Fold_Monad (M : Type -> Type) :=
{
 foldmonad_monad :> Monad M;
 fold : forall X Y,(X->Y->Y)->Y->(M X)->Y
 }.
```

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

```
Class Fold_Monad_Plus (M : Type -> Type) :=
{
  fmonadplus_fmonad :> Fold_Monad M;
  fmonadplus_monadplus :> Monad_Plus M;
  fold_prop : forall X (m : M X),
    fold (fun x=>fun y=>((unit x)[]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).

$$\begin{array}{l}\forall(X : \mathit{Type})\forall(M : \mathit{Tyep} \to \mathit{Type})\forall(m, n : M\ X)\\ \quad \mathit{Fold{-}Monad{-}Plus}\ M \implies\\ \quad\quad \mathit{fold}\ \ f\ \ \mathit{mzero}\ \ (m\Box n)\ \ =\\ \quad\quad (\mathit{fold}\ \ f\ \ \mathit{mzero}\ \ m)\Box(\mathit{fold}\ \ f\ \ \mathit{mzero}\ \ n)\end{array}$$

where $f = \lambda x\ y.(\mathit{return}\ x)\Box y$

A formal proof of this lemma is provided in [11].

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 ϵ-MonadPlus 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 2 ($\Box$-composability). *If M is a carrier for an instance of MonadPlus, and $\psi : Mb \to Ma \to Prop$ is a property of instances of M, then we define $\Box$-composabilty as follows:*

$$\begin{array}{l}\Box\text{–}\mathit{compatible}(Ma, \psi) \overset{\text{def}}{=}\\ \quad \forall j, k : Ma.\\ \quad\quad \forall\langle w_j, \rho_j\rangle : \Sigma n : Mb.\psi(n, j).\\ \quad\quad\quad \forall\langle w_k, \rho_k\rangle : \Sigma n : Mb.\psi(n, k).\\ \quad\quad\quad\quad \psi(w_j\Box w_k, j\Box k)\end{array}$$

Consider the computational content of a proof of $\Box$–*compatible*(Ma, ψ). 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, ρ_j proves $\psi(w_j, j)$ and ρ_k proves $\psi(w_k, k)$ and ρ 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 Γ be a well formed context such that the carrier

$$(M : \mathit{Type} \to \mathit{Type}) \in \Gamma$$

where M is a foldable instance of MonadPlus with compatible membership (ϵ). Also, suppose $(a : \mathit{Type}) \in \Gamma$ and $(b : \mathit{Type}) \in \Gamma$. ϕ is a proposition possibly containing free variables n, m and other variable included in the context Γ. 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 ϕ.

$$\frac{\begin{array}{l}\Gamma, m{:}Ma, \Gamma' \vdash\ \Box\text{–}\mathit{compatible}(Ma, \lambda n\, m.\phi[n, m, \bar{z}])\ \mathbf{ext}\ V\\ \Gamma, \Gamma'[\mathsf{mzero}] \vdash \phi[\mathsf{mzero}, \mathsf{mzero}, \bar{z}])\ \mathbf{ext}\ \rho_z\\ \Gamma, m{:}Ma,\ x{:}a, x \in m, \Gamma'[\mathsf{return}(x)]\\ \qquad\qquad\qquad\qquad \vdash \Sigma n{:}Mb.\ \phi[n, \mathsf{return}(x), \bar{z}]\ \mathbf{ext}\ \langle t_x, \rho_x\rangle\end{array}}{\begin{array}{l}\quad \Gamma, m{:}Ma, \Gamma'[m]\\ \qquad\quad \vdash \Sigma n{:}Mb.\ \phi[n, m, \bar{z}]\ \mathbf{ext}\ \langle m \gg\!= \lambda x.t_x,\ \rho(m)\rangle\end{array}}\ \text{BInd}$$

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

$$\begin{array}{rl}\rho(m) = \pi_3(\mathit{foldr} & \\ & (\lambda\langle k,\ m,\ \rho_k\rangle\langle \bar{k},\ \bar{m},\ \rho_{\bar{k}}\rangle.\\ & \quad \langle k\Box\bar{k}, m\Box\bar{m}, V\ k\ \bar{k}\ \langle m, \rho_k\rangle\ \langle \bar{m}, \rho_{\bar{k}}\rangle\rangle)\\ & \langle \mathsf{mzero}, \mathsf{mzero}, \rho_z\rangle\\ & (\mathit{fmap}(\lambda x.\langle \mathsf{return}(x), t_x, \rho_x\rangle)\ m))\end{array}$$

We prove soundness of rule below in Theorem 15.

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 mzero. Note that Γ' may depend on m so we replace all occurrences of m by mzero there and also in the conclusion $\phi[\mathsf{mzero}, \mathsf{mzero}, \bar{z}]$. The extract of a proof of this hypothesis has the form ρ_z, where ρ_z is a proof term verifying $\phi[\mathsf{mzero}, \mathsf{mzero}, \bar{z}]$. We have eliminated the existential from the antecedent of the sequent because, examining the extract of the conclusion of the rule, we note that $\mathsf{mzero} \gg\!= f = \mathsf{mzero}$ (for any f) and so the extract ρ_z must verify that n is 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 $\mathsf{return}(x)$. The evidence for this is the witness $t_x : Mb$ and a proof term ρ_x which is evidence for $\phi[t_x, \mathsf{return}(x), \bar{z}]$.

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 ϕ.

The instance of *foldr* has the following type.

$$\begin{array}{l} \mathit{foldr} : (\tau \to \tau \to \tau) \to \tau \to M\,\tau \to \tau \\ \text{where } \tau = \Sigma y : Ma.\ \Sigma n : Mb.\, \phi[n, y, \bar{z}] \end{array}$$

Note that the variables $\bar{z}$ will not be free in the context Γ so we will not continue to include them.

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

$$\mathit{fmap}(\lambda x.\langle \mathsf{return}(x), t_x, \rho_x\rangle)\ m$$

The term ρ_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 $\lambda x.\langle \mathsf{return}(x), t_x, \rho_x\rangle)y$ evaluates to a triple $\langle \mathsf{return}(y), t_y, \rho_y\rangle : (\Sigma y : Ma.\Sigma n : Mb, \phi[n, return(y)])$ where $\rho_y : \phi[n, \mathsf{return}(y)]$. Thus, we have used the extract of the proof of hypothesis three (that any monad of the form $\mathsf{return}(x)$ for arbitrary $x \in m$ satisfies the property) to construct evidence for every $y \in m$.

Now consider the fold function itself.

$$(\lambda\langle k,\, m,\, \rho_k\rangle\langle \bar{k},\, \bar{m},\, \rho_{\bar{k}}\rangle.\langle k\Box\bar{k}, m\Box\bar{m},\ V\ k\ \bar{k}\ \langle m, \rho_k\rangle\ \langle \bar{m}, \rho_{\bar{k}}\rangle\rangle)$$

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 $\langle \mathsf{return}(y), t_y, \rho_y\rangle$ and the second is of the form

$$\begin{array}{l} \langle \mathsf{return}(y_1)\Box\cdots\Box\mathsf{return}(y_k), \\ \ \ t_{\mathsf{return}(y_1)\Box\cdots\Box\mathsf{return}(y_k)} \\ \ \ \rho_{(\mathsf{return}(y_1)\Box\cdots\Box\mathsf{return}(y_k))}\rangle. \end{array}$$

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

$$\begin{array}{l} \langle \mathsf{return}(y)\Box\mathsf{return}(y_1)\Box\cdots\Box\mathsf{return}(y_k), \\ \ \ t_y\Box t_{\mathsf{return}(y_1)\Box\cdots\Box\mathsf{return}(y_k)} \\ \ \ \rho_{(\mathsf{return}(y)\Box\mathsf{return}(y_1)\Box\cdots\Box\mathsf{return}(y_k))}\rangle \end{array}$$

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

Theorem 15 (Soundness). *BInd is sound.*

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

$$\begin{array}{l}
(H1)\ \Gamma, m{:}Ma, \Gamma' \vdash V :\ \square\text{–}\mathit{compatible}(Ma, \lambda m.\lambda n.\phi[n, m, \bar{z}])\\
(H2)\ \Gamma, \Gamma'[\mathsf{mzero}] \vdash \rho_z : \phi[\mathsf{mzero}, \mathsf{mzero}, \bar{z}])\\
(H3)\ \Gamma, m{:}Ma,\ x{:}a, x \in m, \Gamma'[\mathsf{return}(x)]\\
\qquad \vdash \langle t_x, \rho_x\rangle : \Sigma n{:}Mb.\ \phi[n, \mathsf{return}(x), \bar{z}]
\end{array}$$

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

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

We proceed by applying lemma 1 (Mplus Decomposition lemma)
There are three cases:

(**case:** $m = \mathsf{mzero}$) In this case we must show

$$\begin{array}{l}
\Gamma, \Gamma'[\mathsf{mzero}]\\
\vdash \langle \mathsf{mzero} \mathrel{>\!\!>\!\!=} \lambda x.t_x, \rho(\mathsf{mzero})\rangle :\\
\qquad\qquad \Sigma n : Mb.\ \phi[\mathsf{mzero} \mathrel{>\!\!>\!\!=} \lambda x.t_x, \mathsf{mzero}, \bar{z}]
\end{array}$$

But we know that, for all f, $\mathsf{mzero} \mathrel{>\!\!>\!\!=} f = \mathsf{mzero}$ so, more simply, we must show

$$\begin{array}{l}
\Gamma, \Gamma'[\mathsf{mzero}]\\
\vdash \langle \mathsf{mzero}, \rho(\mathsf{mzero})\rangle : \Sigma n : Mb.\phi[\mathsf{mzero}, \mathsf{mzero}, \bar{z}]
\end{array}$$

We know $\mathsf{mzero} : Mb$ and so it remains to show that

$$\Gamma, \Gamma'[\mathsf{mzero}] \vdash \rho(\mathsf{mzero}) : \phi[\mathsf{mzero}, \mathsf{mzero}, \bar{z}]$$

By by the rules of *foldr* we have the following:

$$\begin{array}{l}
\rho(\mathsf{mzero}) =\\
\quad \pi_3(\mathit{foldr}\ ...\ \langle \mathsf{mzero}, \rho_z\rangle \mathsf{mzero}) =\\
\quad \pi_3\langle \mathsf{mzero}, \mathsf{mzero}, \rho_z\rangle = \rho_z
\end{array}$$

By (H2) we know $\rho_z : \phi[\mathsf{mzero}, \mathsf{mzero}, \bar{z}]$ which completes the proof of this case.

(**case**: $\exists x : a.\ m = \mathit{return}(x)$) In this case we must show:

$$\begin{array}{l}
\Gamma, x : a,\ \Gamma'[\mathsf{return}(x)]\\
\vdash \langle \mathsf{return}(x) \mathrel{>\!\!>\!\!=} \lambda x.t_x, \rho(\mathsf{return}(x))\rangle :\\
\qquad \Sigma n : Mb.\phi[(\mathsf{return}(x) \mathrel{>\!\!>\!\!=} \lambda x.t_x), \mathsf{return}(x), \bar{z}]
\end{array}$$

Note that $(\mathsf{return}(x) \mathrel{>\!\!>\!\!=} \lambda x.t_x) = (\lambda x.t_x)x = t_x$ so instead we show

$$\begin{array}{l}
\Gamma, x : a,\ \Gamma'[\mathsf{return}(x)]\\
\vdash \langle t_x, \rho(\mathsf{return}(x))\rangle : \Sigma n : Mb.\phi[t_x, \mathsf{return}(x), \bar{z}]
\end{array}$$

By (H3) we know $t_x : Mb$ and so it remains to show the following:

$$\Gamma, x : a,\ \Gamma'[\mathsf{return}(x)] \vdash \rho(\mathsf{return}(x)) : \phi[t_x, \mathsf{return}(x), \bar{z}]$$

By the defintion of ρ,

$$\begin{aligned}&\rho(\mathsf{return}(x)) = \\ &\quad \pi_3(\langle \mathsf{return}(x)\Box\mathsf{mzero}, t_x\Box\mathsf{mzero}, \\ &\quad V\ \mathsf{return}(x)\ \mathsf{mzero}\ \langle t_x, \rho_x\rangle\ \langle \mathsf{mzero}, \rho_z\rangle\rangle) \\ &\quad \pi_3(\langle \mathsf{return}(x), t_x, \rho_x\rangle) = \rho_x\end{aligned}$$

But then, by (H3) this case holds.
(**case**: $\exists(j, k : Ma)\, m = j \Box k$) In this case, we get to assume:

$$\begin{aligned}&(H4)\ \Gamma, j : Ma,\ \Gamma'[j] \\ &\qquad \vdash \langle j >\!\!>= \lambda x.t_x, \rho(j)\rangle : \Sigma n : Mb.\, \phi[n,\ j,\ \bar{z}] \\ &(H5)\ \Gamma, k : Ma,\ \Gamma'[k] \\ &\qquad \vdash \langle k >\!\!>= \lambda x.t_x, \rho(k)\rangle : \Sigma n : Mb.\, \phi[n,\ k,\ \bar{z}]\end{aligned}$$

We must show:

$$\begin{aligned}&\Gamma,\ j, k : Ma,\ \Gamma'[j\Box k] \\ &\ \vdash \langle (j\Box k) >\!\!>= \lambda x.t_x, \rho(j\Box k)\rangle : \Sigma n : Mb.\, \phi[(n,\ j\Box k,\ \bar{z}]\end{aligned}$$

As a result:

$$\begin{aligned}&(H4')\ \Gamma, j : Ma,\ \Gamma'[j] \vdash \rho(j) : \phi[j >\!\!>= \lambda x.t_x,\ j,\ \bar{z}] \\ &(H5')\ \Gamma, k : Ma,\ \Gamma'[k] \vdash \rho(k) : \phi[k >\!\!>= \lambda x.t_x, ,\ k,\ \bar{z}]\end{aligned}$$

By (H1) we know

$$\begin{aligned}&\Gamma, j, k : Ma, \Gamma'[j\Box k] \\ &\ \vdash (V\ j\ k\ \langle t_j, \rho(j)\rangle\ \langle t_k, \rho(k)\rangle) : \Sigma n : Mb.\phi(n, j\Box k, \bar{z})\end{aligned}$$

By mplus–bind property in the definition of MonadPlus we know $\Box$ distributes over bind, thus we have the following:

$$((j\Box k) >\!\!>= \lambda x.t_x) = (j >\!\!>= \lambda x.t_x)\Box(k >\!\!>= \lambda x.t_x)$$

By the induction hypothesis, we know $\rho(j)$ and $\rho(k)$ are evidences of the correct type for j an k. By (H1) $(V\ j\ k\,\langle t_j, \rho(j)\rangle\,\langle t_k, \rho(k)\rangle)$ is evidence for $\phi[(j >\!\!>= \lambda x.t_x)\Box(k >\!\!>= \lambda x.t_x), j\Box k,\ \bar{z}]$.

$\Box$

4 Extracting Monadic Programs

Consider the specification for a function computing all pairs.
$\forall(X, Y : \mathit{Type})\forall(M : \mathit{Type} \to \mathit{Type})\forall(m : M\ X)(p : M\ Y)$
$\exists(n : M\ (X * Y))\ \forall(x : X)\forall(y : Y)\ (x, y)\ \epsilon\ n \leftrightarrow x\ \epsilon\ m \wedge y\ \epsilon\ p$. Assuming that M is a foldable instance of an ϵ-MonadPlus, 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:

$$\begin{array}{l}(G1)\ m\!:\!MX, p\!:\!MY \vdash\ V:\ \Box\text{–}\mathit{compatible}(MX, \lambda m.\lambda n.\phi[n, m, \bar{z}])\\ (G2)\ p\!:\!MY \vdash \phi[\mathsf{mzero}, \mathsf{mzero}, \bar{z}])\\ (G3)\ m\!:\!MX,\ x\!:\!a, x \in m, p\!:\!MY\\ \qquad \vdash \Sigma n\!:\!M(X * Y).\ \phi[n, \mathsf{return}(x), \bar{z}]\end{array}$$

Where ϕ is $\forall(x : X)\forall(y : Y)\ (x, y)\ \epsilon\ n \leftrightarrow x\ \epsilon\ m \wedge y\ \epsilon\ p$.
To prove the first subgoal, by definition of $\Box$-composability, it is enough to show:

$$\begin{array}{l}m\!:\!MX, p\!:\!MY \vdash \forall(j : MX)\forall(k : MX)\\ \quad \forall\langle w_j, \rho_j\rangle : \Sigma(n : M(X * Y))\forall(x : X)\forall(y : Y)\langle x, y\rangle \epsilon n \leftrightarrow x\epsilon j \wedge y\epsilon k\\ \quad \forall\langle w_k, \rho_k\rangle : \Sigma(n : M(X * Y))\forall(x : X)\forall(y : Y)\langle x, y\rangle \epsilon n \leftrightarrow x\epsilon k \wedge y\epsilon p\\ \quad \forall(x : X)\forall(y : Y)\ \langle x, y\rangle\ \epsilon\ w_j\Box w_k \leftrightarrow x\ \epsilon\ j\Box k \wedge y\ \epsilon\ p.\end{array}$$

Now by definition of Σ it suffices to show:

$$\begin{array}{l}m\!:\!MX, p\!:\!MY \vdash \forall(j : MX)\forall(k : MX)\forall(w_j, w_k : M(X * Y))\\ \quad (\forall(x : X)\forall(y : Y)\ \langle x, y\rangle\ \epsilon\ w_j \leftrightarrow x\ \epsilon\ j \wedge y\ \epsilon\ p) \rightarrow\\ \quad (\forall(x : X)\forall(y : Y)\ \langle x, y\rangle\ \epsilon\ w_k \leftrightarrow x\ \epsilon\ k \wedge y\ \epsilon\ p) \rightarrow\\ \quad \forall(x : X)\forall(y : Y)\ \langle x, y\rangle\ \epsilon\ w_j\Box w_k \leftrightarrow x\ \epsilon\ j\Box k \wedge y\ \epsilon\ p.\end{array}$$

Which obviously holds by definition of $\Box$. To prove the second subgoal, we need to show:

$$p\!:\!MY \vdash \forall(x : X)\forall(y : Y)\ \langle x, y\rangle\ \epsilon\ \mathit{mzero} \leftrightarrow x\ \epsilon\ \mathit{mzero} \wedge y\ \epsilon\ \mathit{mzero}.$$

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

$$\begin{array}{l}m\!:\!MX,\ x\!:\!a, x \in m, p\!:\!MY\ \ \vdash\\ \Sigma n\!:\!M(X * Y)\forall(x' : X)\forall(y' : Y)\ \langle x', y'\rangle\ \epsilon\ n \leftrightarrow x'\ \epsilon\ (\mathit{return}\ x) \wedge y\ \epsilon\ p.\end{array}$$

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

$$\begin{array}{l}(G1')m\!:\!MX, p\!:\!MY \vdash\ \Box\text{–}\mathit{compatible}(MX, \lambda m.\lambda n.\phi'[n, m, \bar{z}])\\ (G2')\ \ \vdash \phi'[\mathsf{mzero}, \mathsf{mzero}, \bar{z}])\\ (G3')m\!:\!MX,\ x\!:\!X, x \in m, p\!:\!MY, y\!:\!Y, y \in p \vdash\\ \quad \Sigma n\!:\!M(X * Y)\ \phi'[n, \mathsf{return}(x), \bar{z}]\end{array}$$

Where ϕ' is $\forall(x' : X)\forall(y' : Y)\ \langle x, y\rangle \in n \leftrightarrow x' \in (\mathit{return}\ x) \wedge y \in p$. To prove $G1'$ it suffices to prove:

$$\begin{array}{l}m\!:\!MX, x\!:\!X, x \in m, p\!:\!MY \vdash\\ \ \forall(j : MY)\forall(k : MY)\\ \ \forall\langle w_j, \rho_j\rangle : \Sigma(n : M(X * Y))\forall(x' : X)\forall(y' : Y)\\ \quad \langle x', y'\rangle \in n \leftrightarrow x' \in (\mathit{return}\ x) \wedge y \in j\\ \ \forall\langle w_k, \rho_k\rangle : \Sigma(n : M(X * Y))\forall(x' : X)\forall(y' : Y)\\ \quad \langle x', y'\rangle \in n \leftrightarrow x' \in (\mathit{reurn}\ x) \wedge y \in k\\ \ \forall(x' : X)\forall(y' : Y)\\ \quad \langle x', y'\rangle \in w_j\Box w_k \leftrightarrow x' \in (\mathit{return}\ x) \wedge y' \in j\Box k.\end{array}$$

Which can be proved by proving the following:

$$\begin{array}{l} m\!:\!MX, x\!:\!X, x \in m, p\!:\!MY \vdash \\ \ \forall(j : MY)\forall(k : MY)\forall(w_j, w_k : M(X * Y)) \\ \quad (\forall(x' : X)\forall(y' : Y)\ \langle x', y'\rangle \in w_j \leftrightarrow x' = x \wedge y' \in j) \rightarrow \\ \quad (\forall(x' : X)\forall(y' : Y)\ \langle x', y'\rangle \in w_k \leftrightarrow x' = x \wedge y' \in k) \rightarrow \\ \quad (\forall(x' : X)\forall(y' : Y) \\ \qquad \langle x', y'\rangle \in w_j \Box w_k \leftrightarrow x = x' \wedge y' \in j \Box k) \end{array}$$

Which follows from the definition of $\Box$. The proof of $G'2$ is similar to the proof of $G2$ which uses the fact that nothing belongs to *mzero*. To prove $G'3$ it is enough to show:

$$\begin{array}{l} m\!:\!MX,\ x\!:\!a, x \in m, p\!:\!MY, y\!:\!Y, y \in p\ \ \vdash \\ \ \Sigma n\!:\!M(X * Y)\forall(x' : X)\forall(y' : Y) \\ \quad \langle x', y'\rangle \in n \leftrightarrow x' \in (return\ x) \wedge y' \in (return\ y). \end{array}$$

Since the only element that belongs to $(return\ x)$ is x itself, this is equivalent to:

$$\begin{array}{l} m\!:\!MX,\ x\!:\!a, x \in m, p\!:\!MY, y\!:\!Y, y \in p\ \ \vdash \\ \ \Sigma n\!:\!M(X * Y)\forall(x' : X)\forall(y' : Y) \\ \quad \langle x', y'\rangle \in n \leftrightarrow x' = x \wedge y' = y. \end{array}$$

Which can be proved easily by choosing n to be $(return\langle x, y\rangle)$. The BInd rule has been used twice in this proof. The extraction for the first time is the term $(n >\!\!>\!\!= \lambda y.return\ \langle x, y\rangle)$, and the whole extract is the term $(m >\!\!>\!\!= \lambda x.n >\!\!>\!\!= \lambda y.return\langle x, y\rangle)$.

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 does support a program extraction mechanism [14–16] (even if it is rarely used except as a curiosity). 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 `monad_syn` to inhabitaints 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 `monad_syn`. Such a result is in the folklore, Sculthorpe et. al. [12] state a similar result and cite [13] 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

1. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1) (July 1991) 55–92
2. Wadler, P.: Monads for functional programming. In: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text, London, UK, UK, Springer-Verlag (1995) 24–52
3. Hutton, G., Fulger, D.: Reasoning about effects: Seeing the wood through the trees. In: Proceedings of the Ninth Symposium on Trends in Functional Programming. (2008)
4. Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: ACM SIGPLAN Notices. Volume 46., ACM (2011) 2–14
5. Thompson, S.: Type theory and functional programming. International computer science series. Addison-Wesley (1991)
6. Nordström, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf type theory: an introduction. Clarendon (1990)
7. Sozeau, M., Oury, N.: First-class type classes. In: Theorem Proving in Higher Order Logics. Springer (2008) 278–293
8. Geuvers, H., Pollack, R., Wiedijk, F., Zwanenburg, J.: A constructive algebraic hierarchy in coq. Journal of Symbolic Computation **34**(4) (2002) 271–286
9. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging Mathematical Structures. In Nipkow, T., Urban, C., eds.: Theorem Proving in Higher Order Logics. Volume 5674 of Lecture Notes in Computer Science., Munich, Germany, Springer (2009)
10. Spitters, B., Van Der Weegen, E.: Type classes for mathematics in type theory. Mathematical Structures in Computer Science **21** (7 2011) 795–825
11. Caldwell, J., Shafei, H.: Coq files. `http://www.cs.uwyo.edu/~jlc/papers_chronological.html` (2013)
12. Sculthorpe, N., Bracker, J., Giorgidze, G., Gill, A.: The constrained-monad problem. In: International Conference on Functional Programming, ACM (2013)
13. Apfelmus, H.: The operational monad tutorial. (2010) 37–55
14. Letouzey, P.: A new extraction for Coq. In: Types for proofs and programs. Springer (2003) 200–219
15. Chipala, A.: Certified Programming with Dependent Types. MIT Press (to appear) (February, 13 2013) http://adam.chlipala.net/cpdt/.
16. Pierce, B.C., Casinghino, C., Greenberg, M., Hriţcu, C., Sjoberg, V., Yorgey, B.: Software Foundations. Electronic textbook (2012)