

# Monads through program extraction

Josef Pohl

Department of Computer Science  
University of Wyoming

May 1, 2008

## Abstract

Closing the gap between formalized proof and practical programming has long been an objective of working in type theory. Monads epitomize this balance. They are a formal construct but have revolutionized programming in purely functional environments. We play on this relationship by formalizing the structure of a monad as a type. By instantiating the type constructor and its associated operators in a dependent type theory we are able to reason about monads as we would a type. By the fact that we are working in a constructive context this reasoning means that we can begin to close that gap between a formal proof and practical programming when it comes to monads. We are able to extract instances of the operators by the fact that we can prove propositions that correspond to that type. In this paper we will do this in two varieties. The first is that by knowing the algorithm the operators correspond to we can easily state them as an object in our language. We then prove that they have the correct type, the type of the output for which we are trying to generate. The second methodology is to prove a behavior specification and there by obtain an extract based on what the program is supposed to do. Our extract is in the form of a term in a purely functional programming language that is just as powerful as any more common language. We contrast this with the first method which is a statement of verification, or of how to do a certain computation. In our process of verifying and specifying (extracting) monads we show that particular instances of common monads are definable in our type theory. We show that the monad is a well defined type. And we give examples showing that our monads are usable.

## 1 Introduction

The idea of combining formal verification and programming has been a long sought after goal. One approach to this goal has been in the use of constructive type theory as a foundation for reasoning and proving properties about programs. This has generally taken a theorem proving approach, trying to raise the foundations up into the realm of practical programming. Formal proof systems, such as Nuprl, Coq, Lego, and Alf, have a rich literature developed encompassing this goal. The other approach has generally been realized by programming languages such as Epigram [14], Omega [18], Cayenne [5], and Agda [4],[17] that present a rich dependent type system. What these domains have in common is the idea that we can develop, using the power of the Curry-Howard Isomorphism, a method of programming that combines both proofs and programs as the end result.

In this paper we continue this tradition by using the functional programming language embedded in Nuprl [9] to construct programs. Nuprl is a dependent type theory in the tradition of Martin-Löf. As we describe below, Nuprl's proof system allows for the extraction of programs due to its constructive nature. These programs are terms in a purely functional programming language.

Our motivation comes from working concurrently in Haskell [3] and Nuprl. We have been attempting to find ways of expressing the notion of programming with evidence by using the relationship of constructive membership predicates and the type of indexes into labeled binary trees. [7] Take the typical membership predicate written in Haskell of the form

```
Member :: T * (T tree) -> Bool
Member x t =
  case t of
    Empty -> false
```

```

Node (y, t1, t2) -> x = y
    || Member x t1
    || Member x t2

```

This predicate models the classical notion of membership. But as far as a witnesses go Bool is a rather weak form of evidence. Take, for a moment, a Curry-Howard interpretation of the disjunctions in the Node case, the implicit disjunction implied by the case statement, and the evidence that we may have to provide to show equality in  $x = y$ . We realize that an inhabitant of the Member predicate, or the type, under this interpretation has a much deeper and richer structure than the classical interpretation. In fact the inhabitants of this type are the indexes to the members of the tree. Any evidence for the validity of this statement in the constructive context will say not only that  $x$  is in fact (or is not) in the tree  $t$  but give evidence to this fact in the form of a path or an index into the tree. We modeled this in Haskell and conceived of developing a parallel version formally which would reinforce our notion of programming with evidence. Such a model gives the ability to reason about a global state while staying true to the purely functional environment dictated by our type theory.

## 1.1 Background

Monads provide functional languages with imperative features. Starting from Moggi and Wadler monads have been shown to provide a structure for computing with side-effects, state, IO, and so on without breaking the pure functional nature of the language. There are a large number of tutorials (e.g. [1]) which explain the programming side of monads. And there have been attempts to make the construction of monads systematic namely in [11] and [2]. There has been work in the area of proving monad properties in a formal environment namely in Isabelle/HOL [16]. And much has been done in the use of monads to prove and characterize other programming language features citeleroy, extending the knowledge about monads[10], and combining monads to make them more powerful and useful [13]. This is the first, to our knowledge, to deal with monads by way of program extraction from formalized proof. By defining the components of a monad, and instances of monads, constructively we can use these components to further reason about programs written with monads.

## 1.2 Nuprl as programming environment

Nuprl is a Martin-Löf type theory with extensional function types. Its computation system is the untyped  $\lambda$ -calculus extended to include pairs and disjoint unions. Extracts are composed of these computational terms and are generated from proofs of propositions. Given some type  $T$ , using the proofs-as-programs interpretation to prove a sequent  $\vdash T$  we obtain an extract  $t$ . This  $t$  is exactly a program that has the type  $T$ . In other words the proof of a sequent that has as its conclusion a type  $T$  lays claim to the fact that a witness can be constructed. This witness being a program of type  $T$ . For instance the pair  $\langle a, b \rangle$  is a witness to the proposition (or type)  $A \times B$  when  $A$  and  $B$  are types and  $a \in A$  and  $b \in B$ .

## 1.3 Overview

In this paper we will discuss a formalization of monads in Nuprl's constructive type theory. We give two methods of formalizing monads and give examples of their use. In section 2 we define monads as a type. In section 3 we give an overview of some familiar monads and examples of programs written and verified which use the formalized instance of the monad. Finally we use the full power of the proofs-as-programs paradigm. We specify behaviors that describe monadic based computation and extract functional programs from the proofs of those specifications. The main result of this work is an exposition of monads in a formal environment. In turn we use all the expressive power of a language embedded in a theorem prover to generate programs that use these monads thereby closing the gap between formal environments and practical programming.

## 2 Monads formally defined

The literature explaining the structure of a monad is quite broad starting with [20],[15]. An exposition of a monad generally describes three components required for a monad and three properties that must hold for that combination of structures to be considered a monad. We reiterate these here. <sup>1</sup> A monad is a triple that consists of a type constructor  $M$ , a method to generate something of that type which we call **return**, and a method to compose computations

<sup>1</sup> We note that there is an alternative formulation with *fmap*, *return*, and *join*. We see no obstacles to formalizing this construction as well. For a description of this formulation see [12]

together resulting in another instance of the monad type, `bind` or `>>=`. Their types are as follows:

$M : \mathbb{U} \rightarrow \mathbb{U}$   
 A polymorphic type constructor  
 the type of the monad  
 $return : A \rightarrow MA$   
 A polymorphic function constructing something  
 of type M  
 $>>= : M A \rightarrow (A \rightarrow M B) \rightarrow M B$   
 A method of applying a function to a monad,  
 and hence composing monad functions.

This instance of a polymorphic type constructor is a function from types to types as will be described further shortly .

## 2.1 Laws

Monads will additionally adhere to three laws. We need to prove that `>>=` is associative. It also needs to be shown that `return` acts as both a right and left identity with `>>=`. These theorems are stated formally below. Although they are generally only used in this formal context it should be noted that it is an obligation of the programmer to be assured of these properties when instantiating a monad. We take `>>=` to be an infix operator,  $a$  to be in the set of value variables, and  $A$  as an arbitrary type , reserving  $m$  to be an instance of a monad,  $m : (M A)$ .

Left identity  
 $return\ a\ >>= f = fa$   
 Right identity  
 $m\ >>= return = m$   
 Associative  
 $(m\ >>= p)\ >>= q =$   
 $m\ >>= (\lambda x.(p\ x)\ >>= q)$

In ordinary programming the monad operator laws rarely come into play if at all. The instance of a monad is programmed directly in Haskell, for instance. However, in our exposition they are crucial to the development as we use them to formulate a well-defined type, providing explicit evidence to the correctness of the monad and the definition of the type.

## 2.2 Types and Computational Terms in Nuprl

As noted above, Nuprl is a dependent type theory in the style of Martin-Löf. It has extensional function equality and type universes,  $\mathbb{U}_i$ , where  $i$  is a level in

the hierarchy of type universes. We generally will work at an arbitrary type level  $i$  and denote this universe as  $\mathbb{U}$ . If  $A$  is a type then it is a member of a type universe. We use  $A : \mathbb{U}$  to designate  $A$  is a type, and  $A \in \mathbb{U}$  as an obligation that  $A$  must be shown to be a well-formed type. Generally some element  $x$  is denoted as a member (an element or term) of type  $A$  by  $x : A$  and reserve  $x \in A$  as a well-formedness obligation as before.

We subscribe to the notion of proposition-as-types. This is a correspondence that states, for any proposition,  $P$  there is a corresponding type  $\bar{P}$ . We denote propositions  $\mathbb{P}$  and state  $P : \mathbb{P}$ . Further we equate propositions and types,  $\mathbb{P} = \mathbb{U}$ .<sup>2</sup> Thus, a proof of that proposition shows the corresponding type to be inhabited, or there exists an element of that type. It will come into play below that if we have a proof of a proposition  $P$  we know we can find a witness. This witness is the fact that there is a  $\bar{p}$  that inhabits the corresponding type  $\bar{P}$ . We will use this fact explicitly in specifying and extracting programs that are exactly the `return` and `bind` from above. (We use `bind` in our formalization mostly for clarity and readability of the resulting definitions, theorems, and tactics. Any place `bind` is found, `>>=` can be substituted. )

Again, extracts in Nuprl are computational terms and the computational language is equivalent to one in a language such as Haskell. The terms and types in Nuprl are closely related. Terms can be seen as computable expressions when they have been associated with, or generated from, a well-defined type. Further they can be evaluated just as any function given in Haskell. Unlike a number of other type theories, Nuprl's constructive type theory allows construction and computation of terms using general recursion. We will point out a few of the type constructors and the terms associated with them here. We note that just as one can build a program from component terms in Haskell we can build a program in Nuprl from our term language, or extract it from a proof that a type is inhabited.

If we assume  $A$  and  $B$  to be well-defined types then we can build new types from these component types. The simple function type is designated as  $A \rightarrow B$ . Inhabitants of this type are functions which transform an element of type  $A$  into an element of type  $B$ . To prove a theorem of this type  $\vdash A \rightarrow B$  we provide a witness which is the function itself.

The disjoint union of two types is denoted  $A + B$ .

<sup>2</sup>This is different than Coq where types and propositions reside in different hierarchies.

This type is inhabited by elements of the form  $inl(a)$  and  $inr(b)$  where  $a : A$ ,  $b : B$  and  $inl(t)$  and  $inr(t)$  are constructors for the disjoint union type. The type corresponds to the proposition  $A \vee B$ . Proving a type  $A + B$  is inhabited is equivalent to showing that either  $A$  or  $B$  is inhabited along with the identification of which one has been shown. The general form of the destructor for Disjoint Union is a decide term.

$$Decide(t, x.t_1(x), y.t_2(y))$$

The computational rules of disjoint union act much like a case analysis and are given as follows.

$$\begin{aligned} Decide(inl(t), x.t_1(x), y.t_2(y)) &\rightarrow t_1(t) \\ Decide(inr(t), x.t_1(x), y.t_2(y)) &\rightarrow t_2(t) \end{aligned}$$

The variables  $x$  and  $y$  in  $x.t_1(x)$  and  $y.t_2(y)$  are bound instances. Informally we consider the disjoint union as tagging elements as being from either  $A$  or  $B$ . Hence the extract of a proof of a disjoint union is an operation on cases.

The Cartesian Product, or simply product, type is the type of pairs.  $A \times B$  is inhabited by elements  $\langle a, b \rangle$  where  $a : A$  and  $b : B$ . We define two destructors,  $\pi_1$  and  $\pi_2$ , called projection operators. Computationally these are defined as:

$$\pi_1 \langle a, b \rangle \rightarrow a \quad \pi_2 \langle a, b \rangle \rightarrow b$$

We can nest pairs as well as projections to create larger tuples and extract information contained within them.

We also introduce the set type with comprehensions. This is noted as

$$\{x : A \mid B(x)\}$$

We think of it as a subtype of  $A$  determined by the property  $B(x)$ . We read it as  $a \in A$  such that  $B(a)$ . The value in this is that if we prove a fact  $\vdash \{x : A \mid B(x)\}$  we are left with only the witness of  $x$  without the additional evidence of  $B(x)$ . In most cases proving a proposition results in the extract, the computational content, and the evidence or the proof of the fact. With the set type we hide the evidence and are left with the program.

Finally a type is discrete if equality is decidable for that type.

**Definition 1 (discrete)**

$$discrete\ T \stackrel{\text{def}}{=} \forall x, y : T. x =_T y \vee \neg(x =_T y)$$

This is also often denoted as the proposition  $Decidable(T)$ . For instance, natural numbers are decidable, but the type of functions from  $\mathbb{N}$  to  $\mathbb{N}$  ( $\mathbb{N} \rightarrow \mathbb{N}$ ) are not decidable.

## 2.3 Monads as a Nuprl Type

In order to extract programs which use monads we define a type of monads. The first step in this process is to show that we can construct a type of monads from simpler components, and prove that it is well typed. To do this we consider the definition of a monad, namely the type constructor  $\mathbb{M}$  and the `bind` and `return` operators. This will define the core of our monad. We note that our monad type not only contains type constructors and operators, but must adhere to the monad laws. A formalization of a type of a monad would be incomplete without an assurance that the monad is completely defined. Hence our monad is 6-tuple containing the type constructor, the operators, and what amounts to three theorems stating the right and left identity and associativity hold for the aforementioned constructors and operators. We can formally instantiate our monad type as a 6-tuple.

**Definition 2 (Monad)**

$$\begin{aligned} Monad(\mathbb{U}) = & \\ M : \mathbb{U} \rightarrow \mathbb{U} \times & \\ \forall A, B : \mathbb{U}. bind : M\ A \rightarrow (A \rightarrow M\ B) \rightarrow M\ B \times & \\ \forall A : \mathbb{U}. return : A \rightarrow M\ A \times & \\ associative(bind) \times & \\ right\_identity(bind, return) \times & \\ left\_identity(bind, return) & \end{aligned}$$

A monad is any instance of this type. In other words if we instantiate a  $M$ , a  $bind$ , and a  $return$  along with the proofs that these components together satisfy the monad laws, we have an instance of a monad, or more exactly the monad type. Further it defines a *Monad* as a dependent type, specifically a dependent product type, where the laws, *associative*, *right\_identity*, and *left\_identity* are predicates that depend on their respective arguments. The proof obligation here is to show, component-wise, that each piece is in fact a well-formed type. We define each component as an abstraction in Nuprl's term language.

We also note that we would be unable to prove that a monad is a type in the same universe as its component pieces are. Hence we look to prove the following theorem to state formally that the above definition is a type.

**Theorem 1**

$$Monad(\mathbb{U}_i) \in \mathbb{U}_{i+1}$$

This states that any monad whose components are defined at one level of the hierarchy is actually a type

in the next level up. From a pragmatic standpoint this increase in the hierarchy does not concern us but it is a concern in the type theory. The proof of this theorem consists of showing that each component is in fact a type as well. It is often the case that we will need to use portions of the monad type or a defined instance. In these cases we can use our destructors  $\pi_1$  and  $\pi_2$  to extricate the component we need from the instance of a monad. Further we can define them as abstractions, named components referring to their location in the product. For example, if we have a monad instance  $M_{inst} : Monad(\mathbb{U}_i)$  we can define the following abstractions.

$$M \stackrel{\text{def}}{=} \pi_1(M_{inst})$$

$$bind\_M \stackrel{\text{def}}{=} \pi_1\pi_2(M_{inst}) \quad return\_M \stackrel{\text{def}}{=} \pi_1\pi_2\pi_2(M_{inst})$$

In what follows we will, when speaking about a specific instance, refer to a component by its name followed by the monad instance type as above. If we are speaking generically we will just refer to the name of the component knowing that if it is part of a monad we can also refer to it in terms of the product. The instances of the monad laws can be referred to similarly.

Below we describe, briefly, the proof of each component. This is actually done in the proof that  $Monad(\mathbb{U}_i)$  is a type. However we show them separately for clarity.

We are initially concerned with the first three components of the monad. We define the type constructor to be a type itself. In particular it must be a function from types to types.<sup>3</sup> Hence we show that:

**Lemma 1**

$$M \in \mathbb{U} \rightarrow \mathbb{U}$$

$M$  is an arbitrarily defined typed the proof results from the definition. The `bind` and `return` operators are similarly formalized as function types. Here, however, we prove that `bind` and `return` inhabit particular instances of function types asserting the existence of those component types  $A$  and  $B$ . We prove the following two lemmas:

**Lemma 2**

$$bind \in \forall A, B : \mathbb{U}. M A \rightarrow (A \rightarrow M B) \rightarrow (M B)$$

**Lemma 3**

$$return \in \forall A : \mathbb{U}. (A \rightarrow M A)$$

<sup>3</sup> This is exactly the reason that a monad must be shown to be a type in a higher level of the hierarchy of types.  $\mathbb{U}_i \rightarrow \mathbb{U}_i : \mathbb{U}_{i+1}$

Having defined an abstract version of these types we are able to then show that a triple of type constructor and the two operators that match these types may adhere to the three monad laws. Hence we prove that the operators have the property of left and right identity and associativity. The instance of `bind` and `return` in the follow lemmas are variables which can be instantiated, as we will see, with the actual definitions of the operators we create.

**Lemma 4**

**Associative :**

$$\forall A, B, C : \mathbb{U}$$

$$\forall p : (M A).$$

$$\forall q : (A \rightarrow (M B)).$$

$$\forall r : (B \rightarrow (M C)).$$

$$(p \text{ bind } (A B) q) \text{ bind } (B C) r =$$

$$p \text{ bind } (A B) (\lambda x. (q x) \text{ bind } (B C) r)$$

$$\in M C$$

**Lemma 5**

**RightIdentity :**

$$\forall A : \mathbb{U}$$

$$\forall p : (M A).$$

$$p \text{ bind } (A A) \text{ return } A = p$$

$$\in M A$$

**Lemma 6**

**LeftIdentity :**

$$\forall A, B : \mathbb{U}$$

$$\forall p : (A \rightarrow M B).$$

$$\forall x : A.$$

$$(return A x) \text{ bind } (A B) p = p x$$

$$\in M B$$

The first 3 lemmas are well-formedness theorems. They are shown explicitly since typing in Nuprl is undecidable, hence we have the obligation to show that our abstractions or definitions are well-typed (or formed). The types parameterize the operators making the syntax a bit unwieldy at times. This is a consequence of the obligation to show that the operators are well-typed in the context of other types. We can define display forms that would hide this information. The obligation will remain that, for instance, if we instantiate an instance of `return`, it must be shown to be a function from type  $A$  to type  $M A$ . There is no valuable computational content in a well-formedness goal as membership in a type equates to equality within that type and is inhabited only by `.` (pronounced 'it') or *Axiom*. The computational content will come from the instantiations of functions that have these types.

The last 3 lemmas are actually propositions. We are stating that the two terms, the left and right hand side, are equal in some type. But we note again the equivalence of propositions and types. Any proof of  $A \in \mathbb{U}$  is in turn proof of the equivalent  $A \in \mathbb{P}$

We now have the framework to show specific instances of the monad type. We can do this by defining the components piecemeal and build the instance of the monad type. Or the proof of the instance can be defined all at once as a 6-tuple and shown to be an instance of the monad type.

### 3 Examples

There have been a large number of tutorials giving examples of easily understood and rather useful monads. We formalize some of those examples here. Defining each instance of a monad follows a common methodology. First the components are defined, then we prove the laws, and finally show the monad type is satisfied. The largest issue, as might be expected, is correctly specifying the types. As noted above, the type parameters in the definition of the operators are necessary to prove that they do in fact meet the constraints of being a monad. We first give a concrete example of the  $ID$  monad and the discuss the general process.

#### 3.1 Id

Most tutorials on monads start with an exposition of the Id monad, and we do the same. Our id monad,  $ID$ , has as its type constructor the identity function,  $\lambda A.A$ . Hence we define and prove the following:

**Definition 3 (ID type constructor)**

$$ID == \lambda A.A$$

**Theorem 2**

$$ID \in \mathbb{U} \rightarrow \mathbb{U}$$

Similarly we define the operators **bind** and **return** as function types. In this case **return** is the application of the type constructor itself and **bind** is standard composition.

**Definition 4 (return\_ID)**

$$return\_ID == ID$$

**Theorem 3**

$$\forall A : \mathbb{U}. return\_ID \in A \rightarrow (ID A)$$

**Definition 5 (bind\_ID)**

$$m \text{ bind\_ID } (A B) p = p m$$

**Theorem 4**

$$\begin{aligned} &\forall A, B : \mathbb{U}. \forall m : (ID A). \\ &\forall p : (A \rightarrow ID B). m \text{ bind\_ID } p \in (ID B) \end{aligned}$$

In the latter case we are required only to prove that the instantiation has the result type of  $ID B$  on the assumption of an appropriate  $m$  and  $p$ .

Once these obligations have been met, in order to fully accept  $ID$ , **bind\_ID**, and **return\_ID** as an instance of a monad we must show that it holds to the monad laws. Hence we prove the following:

**Lemma 7**

**Associativity of ID**

$$\begin{aligned} &\forall A, B, C : \mathbb{U}. \\ &\forall m : ID A. \\ &\forall q : A \rightarrow (ID B). \\ &\forall r : B \rightarrow (ID C). \\ &m \text{ bind\_ID } (A B) q \text{ bind\_ID } (B C) r = \\ &m \text{ bind\_ID } (A C) (\lambda x.(q x) \text{ bind\_ID } (B C) r) \\ &\in ID C \end{aligned}$$

**Lemma 8**

**Right Identity of ID**

$$\begin{aligned} &\forall A : \mathbb{U}. \\ &\forall m : ID A. \\ &m \text{ bind\_ID } (A A) return\_ID = m \\ &\in ID A \end{aligned}$$

and finally

**Lemma 9**

**Left Identity of ID**

$$\begin{aligned} &\forall A, B : \mathbb{U}. \\ &\forall p : A \rightarrow ID A. \\ &return\_ID \text{ bind\_ID } (A B) p = p \\ &\in ID B \end{aligned}$$

Having these theorems then allows us to show that  $ID$  along with the associated laws and operators forms a monad. We state and prove this formally.

**Theorem 5 ID is a Monad**

$$\begin{aligned} &\langle ID, \\ &(\lambda A, B, m, p. m \text{ bind\_ID } (A B) p), \\ &(\lambda A, a. return\_ID a), \\ &extract\_of(Associativity \text{ of } ID), \\ &extract\_of(Right \text{ Identity of } ID), \\ &extract\_of(Left \text{ Identity of } ID) \rangle \in Monad(\mathbb{U}) \end{aligned}$$

The proof is straight forward. We prove, in turn, that the first three components match the type as specified in the original Monad type. This is accomplished by essentially unfolding the definitions of ID and Monad and using a number of reduction and rewrite techniques and tactics built into Nuprl. The laws are shown to be of the right type (propositions) by a similar process of reduction and unfolding.

The item of interest in this theorem is not the proof itself but it is in the use of the extract of the lemmas concerning associativity and the left and right identities. The extract of these theorems can be thought of informally as a function whose sole output is a statement of validity or equality between the left and right hand side. The extract of the lemmas is a slightly different use of the extraction mechanism. Here we are stating that a witness can be found that to the validity of the proposition. Had we instead stated the theorem as an implication, recalling once again the correspondence between implication and functions, we would have had as an extract a method, a function, by which to compute the association or identities. This method and use of extraction will come into play shortly, but in the current situation it is sufficient to use the equality formulation with the extract being *Axiom*.

In the remainder of this section we will forgo the detailed development of each theorem for each monad. The details are quite similar. The process and proof are the same modulo the amount of reduction necessary and how we handle certain unique aspects. Hence we give the formal characterization of the type constructor and note where the formalization differs.

### 3.2 Discussion

We first consider the type constructor. A type constructor in our formalization is a function from types to types ( $\mathbb{U} \rightarrow \mathbb{U}$ ). For instance we can define a type of labeled trees as being a function that takes a type parameter and returns a type. Hence, for any monad type constructor we must first show that it is a function.

The operators are given by abstractions where we give the definition of computation for that instance. The **return** operator is similarly a simple matter. We define an abstraction that takes not only the type information but an element of that type to construct an instance of that monad type. Confusion may arise in the over loading of the **return** function as to whether it is constructing a type or an inhabitant of that type. It is clarified by the use of the function. **Return**, too, is defined as a function type in terms of the type con-

structor. The **bind** operator becomes more complex as the methodology of composing computations is embedded in the definition. Here we will often instantiate an arbitrary instance of the monad,  $m : M A$ , and some function  $p : A \rightarrow M B$  for use in the definition of **bind**. In both cases, as in a definition in a language such as Haskell, the actual implementation is unique to the monad being defined.

With both operators and the type constructor, if we are to instantiate an instance of them as a monad, we are obligated to show that the 6-tuple is of the monad type. This is a process of showing that for each component we can prove that component has the type of the corresponding component in the monad type. Hence for a specific instance or definition of **return**,  $returnM$ , we would state the well-formedness theorem:

$$\forall A : \mathbb{U}. returnM \in A \rightarrow M A$$

and for  $bindM$

$$\forall A, B : \mathbb{U}. bindM \in A \rightarrow (A \rightarrow M B) \rightarrow M B$$

and a similar theorem for the type constructor  $M$  to show that it is in  $\mathbb{U} \rightarrow \mathbb{U}$ . These obligations are generally straightforward proofs.

In the remainder of this section we give brief expositions of instances of some of the monads  $M$  we have formalized. We have, with the exception of the additional but necessary typing information, stayed true to the traditional definitions of each monad instance.

### 3.3 Maybe

One of the more common and useful monads for computation is the Maybe monad. This monad encapsulates computations that may not return a result. Nuprl does not, in its core libraries contain such a type so it is up to us to defined it explicitly for this task. We define the type of Maybe in terms of the disjoint union of the *Unit* type (the type with a single element,  $(\bullet)$ ) and the type  $A$ . We can differentiate the two by choice of the left or the right disjunct. This is our equivalent to pattern matching. Maybe is defined as follows.

$$Maybe == \lambda A. Unit + A$$

We can define *Just a* and *Nothing* using the cases  $inr(x)$  and  $inl(\bullet)$ , respectively.

$$Nothing == inl(\bullet)$$

$$Just x == inr(x)$$

And of course we can prove that both are in the type *Maybe A*.

The development, as noted, follows a similar vein. Our proofs are facilitated by meta-programs developed in Nuprl to do the case analysis as needed. *Maybe* as a type constructor is easily discharged to be of the type  $\mathbb{U} \rightarrow \mathbb{U}$  and the remainder of the operators and lemmas are defined and shown easily. The *bind\_Maybe* operator is defined on case analysis.

$$m \text{ bind\_Maybe } (A B) p == \begin{array}{l} \text{case } m \text{ of} \\ \text{Nothing} \rightarrow \text{Nothing} \\ \text{Just } a \rightarrow p a \end{array}$$

And *return\_Maybe* invokes the *Just* constructor.

$$\text{return\_Maybe} == \lambda a. \text{Just } a$$

The three laws are discharged by computation (reduction and unfolding). *Maybe* is easily shown to be a monad, per our earlier development.

### 3.4 List

Lists in Nuprl are a primitive datatype.<sup>4</sup> We exploit this in proving properties about our monad type constructor and operators for the List Monad. However we note that although List is defined to be a parametrized datatype. It is not defined as the type  $\mathbb{U} \rightarrow \mathbb{U}$ . Hence we have to manipulate it in such a way that it can be accepted as our monad type constructor.

$$\text{ListM} == \lambda A. A \text{ List}$$

where *ListM* is our monad type constructor and *A List* is the Nuprl list type parametrized by type *A*. The *bind\_ListM* and *return\_ListM* functions are defined as would be expected but are stated in terms of *ListM* rather than *A List*. In any theorem and subsequent proof we can unfold the definition of *ListM* and reduce the result to obtain a proof about lists.

$$m \text{ bind\_ListM } (A B) p == \text{concatMap}(p, m)$$

where *m* is a *ListM A* and *p* is a function  $A \rightarrow \text{ListM } A$ . and

$$\text{return\_ListM} == \lambda x. [x]$$

Once again the proof that *ListM* is a monad is straight forward.

<sup>4</sup>Lists are defined as a base type in the lisp code which defines Nuprl.

### 3.5 State

Our most interesting and complex example is of the State monad. The issue here is that *State* itself is parametrized over a type (that of the type of, or the structure of, the state). We simplify matters by choosing a particular form for the actual state of the computation. The container we choose to hold our state is a list parametrized over type *S*. However we could have chosen an arbitrary type *T* to be determined when we actually instantiated the State monad in a program. State is a function from the container holding the state to a pair of a new container and another type.

$$\text{State } S == \lambda B. S \text{List} \rightarrow (S \text{List} \times B)$$

With this definition *State S* is easily seen to be a type from  $\mathbb{U} \rightarrow \mathbb{U}$ . The operator *return\_State* is defined as a function that results in a pair.

$$\text{return\_State } S = \lambda b. l.(l, b)$$

The well-formedness theorem is stated as follows.

#### Lemma 10

$$\forall S, B : \mathbb{U}. \text{return\_State } S \in B \rightarrow \text{State } S B$$

It is easily discharged by unfolding the definition of *State*.

The definition of *Bind\_State* is based on several examples of it in the literature. The well-formedness proof again relies on unfolding of the definition and reducing.

$$m \text{ bind\_State } (S B) p == \begin{array}{l} \lambda x. \text{let } \langle l, b \rangle = (m x) \text{ in} \\ \text{let } st1 = (p b) \text{ in} \\ (st1 (l)) \end{array}$$

The statements of the definitions, theorems, and the proofs become complicated only in the carrying of extra types around and discharging them as needed.

### 3.6 Examples with Trees

As we are motivating the use of a formal theorem proving environment as a programming environment we will give a few examples of programs developed in Nuprl. We should note that we are using a typical methodology of verifying, proving, that our results have the correct type. In these cases we do actually have the algorithm or function that we are trying to develop. The resulting extract will contain only the computational content of the proof. Which is to

say that it is a proof of  $x \in T$  or  $x$  is an element of type  $T$ . Here are examples of the process of using our formalized monads to state two basic functions found in [6]. The first using the ID monad and the second, a more in depth example, using the State monad.

### 3.7 SumTree

The idea behind SumTree is to simply accumulate all of the values stored in a tree. It exemplifies how we verify a monadic program. The algorithm is a traversal of a binary tree labeled with natural numbers. The value at each node in the tree is collected by summing the nodes value with all that have been encountered so far. Nuprl's *letrec* operator defines the procedure of making recursive calls in our function. We do case analysis on trees parametrized by natural numbers. We define our trees in a more programmatic fashion for readability.

$$Tree \mathbb{N} \stackrel{\text{def}}{=} Empty | Node(\mathbb{N}, Tree \mathbb{N}, Tree \mathbb{N})$$

We use the ID monad in this case to collect the summed value and report it at the end of computation through the tree. *SumTree* is defined as follows.

```
sumTree (t) =
(letrec s(t) =
  case t:
    empty → (return_ID 0)
    node(x,t1, t2) →
      (return_ID x )
        bind_ID (N N)
          (λ y. (s t1)
            bind_ID (N N)
              (λ n. (s t2)
                bind_ID (N N)
                  (λ m. return_ID
                    (y + n + m)))) ) t
```

We can prove a theorem that shows exactly what we would expect, that *sumTree* results in a value,  $ID \mathbb{N}$ , which is just  $\mathbb{N}$  by the definition of *ID*.

$$\forall t : Tree \mathbb{N}. sumTree(t) \in ID \mathbb{N}$$

We can either use the statement *sumTree* in Nuprl's computational environment or prove a theorem with the tree,  $t$ .

```
t = Node(2,
  Node(3, Empty, Empty),
  Node(4, Empty, Empty))
```

### Theorem 6

$$sumTree t = ID 9$$

The proof is done by unfolding *sumTree* and reducing it to the value of 9. It reduces completely to a natural number. (We could have stopped computation at the term *ID9*.

$$\begin{aligned} ID(ID 2+ \\ ID(ID 3 + ID 0 + ID 0)+ \\ ID(ID 4 + ID 0 + ID 0)) \\ \Rightarrow ID9 \Rightarrow 9 \end{aligned}$$

A trivial example indeed but again it exemplifies the use of Nuprl's computation system.

### 3.8 NumberTree

We now present a slightly more complex example. *NumberTree* is a function that uses a global state to store the values seen in a traversal. It stores them in the order in which they were first encountered in the tree. Our formulation of state as a list from section 3.6 allows a convenient way to store this information. We need to define additional "helper" functions to facilitate our computation. For instance *numberNode*, *nNode*, and *lookup* are functions that we use in the definition of *numberTree*. Our definition of *numberTree* is dependent on having a discrete equality, *eq*, on the type  $T$  stored in the tree under evaluation. This information is passed along as arguments. *NumberTree* produces two artifacts at each step of the traversal. First a *State* which contains a list of elements of type  $T$ . And secondly a new tree of type  $Tree \mathbb{N}$ . The value at the node in the new tree is the location where the element  $t'$  occurred in the state, or the size of the state if  $t'$  is a new value.

```
numberTree T t ==
  (letrec numT t T =
    case t:
      empty →
        (return_State (Tree N) empty)
      node(x,t1 , t2) →
        numberNode x T
          bind_State (N (Tree N))
            λ num. numT (t1) T
              bind_State ((Tree N) (Tree N))
                λ nT1. numT (t2) T
                  bind_State ((Tree N) (Tree N))
                    λ nT2. (return_State (Tree N)
                      Node(num,nT1,nT2))) T t
```

Stating the definition of `numberTree` in Nuprl is not much more difficult than writing it in a functional language such as Haskell. However the types add a level of complexity, but a necessary complication. The main concern in this situation is that `numberTree` results in a value of the correct type. In this case it does and the value is shown to be a State Monad.

**Theorem 7**

$$\begin{aligned} &\forall T : \mathbb{U}. \\ &\quad \forall t : (Tree\ T). \\ &\quad Decidable(T) \Rightarrow \\ &\quad \quad numberTree\ T\ t \in State\ T\ (Tree\ \mathbb{N}) \end{aligned}$$

Proving correctness (or verifying) of a function in the type theory implies that we can show not only that the function has the correct type but the types are properly applied. For instance in `State T (Tree N)`, `T` is the type held in the state while `Tree N` is the type the monad is parametrized over. At each step of the proof we are obligated to show that the types are well formed but are applied correctly. The above theorem tells us that it has the expected type. It should be noted that a discrete equality on the type `T` is needed, hence `Decidable(T)`. It is hidden in the display of `numberTree` Finally by specifying an extraction program, `extract s`, the resulting of type `Tree N` could be retrieved. The composition of `extract o numberTree T t` is of the type `TreeN`.

We have tried to keep all the practical features of programming with monads: modularity, flexibility, and isolation. By the above exposition we add in the benefit of formal correctness. Unfortunately the strength of the type system adds in overhead. Much of this can be abstracted away in an implementation.

## 4 Specification of bind and return

Given a proof of  $\vdash P$  where  $P$  is some proposition. We give  $\vdash P$  a name, *PTHM*. From this proof we can then create an extract, or synthesize a program,  $p$ . We designate this as  $\vdash P \text{ ext } p$ . When instantiating it in a theorem we will write `ext {PTHM}`. The methodology we used in the previous sections was to state explicitly,  $\vdash p \in P$ . Or in other words we prove  $p$  has the type  $P$ . The program  $p$  is an implementation in the functional programming language we described earlier. We note that *if*  $\vdash (P \text{ ext } p)$  then  $\vdash (p \in P)$ . We can, given an extract  $p$  always prove that it has the correct type.

The other way, to generate the proposition from an instance, is not as easy without using  $p$  as a witness in the proof. The membership proof can facilitate the proof of  $\vdash P$  but there is no direct translation.

We now depart from our methodology of verifying functions that we know how to write and approach monad development from the angle of specification. Previously we had stated in our theorem prover the definition of a function, for instance `sumTree`, or `bind_ID`. The rationale for this is that we know exactly how we are supposed to compute something. But let's approach the same problem from the notion that we may not know how to do something but we do know what we want to do. From this angle we will see that we can build an equivalent monad not from implementations that we state explicitly but from a specification of the behavior of the monad. We still need, after having specified the behavior of our operators, to assert that this behavior models a monad. Or in other words that the components fit into our 6-tuple. The components that we insert now are not the definitions of functions that we wrote down but the extracts of theorems that describe that behavior.

Our goal here is to give a few examples of specifying `bind` and `return`. These are relatively well defined functions that have a well specified behavior. For instance the `bind` operator for the ID monad is normal function application. Hence in most of these cases it is more than likely easier to state the operator as a term. However if our more lofty goal is to embed specified monadic features such as state and other side-effects into programs, or the specification of programs that we are trying to extract from the proofs, we are on the right track.

### 4.1 Working with extracts

Working with extracts in Nuprl, contrary to their discovery and creation, is not necessarily difficult. We instantiate an extract of one theorem into the statement of another theorem by presenting it as a term. For instance if we proved a theorem with the name `int_SqRt` whose extract was a function generating the integer square root of a number we could instantiate that extract in another theorem.

$$\exists n : \mathbb{N}. (\text{ext}\{int\_SqRt\}\ 10) = n$$

The witness to this trivial theorem would be 3. But the power comes from the fact that we can use the extract as a function. A function that was embedded in the proof of `int_SqRt`. We unfold the extract `ext{int_SqRt}` in place and proceed to use the func-

tion as if we had explicitly stated it as a defined object in our term language.

In what follows we will see that we do exactly as described above for `bind` and `return`. We redevelop two monads by specifying the operators and using their extract to rebuild our monads. In this development we assume that we have the type constructor that we are looking to build the monad around. The reason for this is that the types themselves, in most of these instances, are constructed from other types via the constructors from above or are predefined in our computational language. We would gain little from the experience of specifying them, if we could in all cases. There are exceptions to this and we shall continue to explore more complicated types and monads along these lines.

## 4.2 Specifying return and bind

Specifying the `return` is often no more complicated than showing that it would be possible to find a witness to the following general proposition.

$$\forall A : \mathbb{U}. M A$$

The witness, or the extract, in this case is exactly a function that takes an arbitrary type  $A$  and builds an instance of type  $M A$ . Creating such a function is not usually difficult but `return` must be defined to interact properly with `bind` so the monad laws to be provable.

The `bind` operator is generally more complex. Specifying the behavior of `bind` can be a process of specifying the individual behavior of each of the parts of the operators body. For instance in an error monad the `bind` operator can be stated as:

```
data M a = Raise Exception | Return a
type Exception = String
m >>= k =
  case m of
    Raise e → Raise e
    Return a → k a
```

This can be specified as a disjunction of two propositions. Thereby emulating the pattern matching. We in essence generate two proofs, or two programs. One generating the left disjunct, the identity function when the term matches `Raise e`, or `inl (e)`, as it would be in our computational language. And a second program relating `Return a` with the application of `k a`.

$$\forall a : A. Return a \Rightarrow k a$$

This combination of proofs covers a full description of the behavior and we can think of it as two separate functions that each compute half of the conjunct. But in reality it is one proof and one extract.

## 4.3 Extracting the ID monad

Restating the ID monad is relatively similar to before. Specifications of the behavior of `return` and `bind` for the  $ID$  monad are propositions describing the identity function and standard function application. The reasoning involved is minimal. However to match our original verified monad and to use our monad type framework we must again take special consideration of the types. We state and prove the theorem for  $ID\_spec\_return$  and  $ID\_spec\_bind$ . In both these cases we prove the existence of a witness to the fact that the type in question is inhabited. Hence we are proving  $\vdash P$ .

### Theorem 8 ID\_spec\_return

$$\forall A : \mathbb{U}. \forall x : A. ID A$$

The question may arise, what is the additional  $x$ ? We note that we need a witness to  $A$ . This witness, or assertion, if it exists allows us to discharge the proof of  $A : \mathbb{U}, x : A \vdash ID A$ . Here we are not trying to prove that  $ID A$  is a type we are constructing an element of type  $ID A$ . This theorem provides us with an extract that is exactly `return_ID` from our original formulation. We can prove this equality by stating that the abstraction we used in our proof that ID was a monad is equivalent to the extract when applied to type  $A$ .

### Theorem 9 return\_ID extensionally equal

$$\forall A : \mathbb{U}. (\lambda A, a. return\_ID a) A = \{\text{ext}\} ID\_spec\_return A \in A \rightarrow ID A$$

`Bind_ID` has a similar specification.

### Theorem 10 ID\_spec\_bind

$$\begin{aligned} \forall A, B : \mathbb{U}. \\ \forall i : ID A. \\ \forall f : A \rightarrow ID B. \\ ID B \end{aligned}$$

This theorem states that  $ID B$  is inhabited assuming  $A, B, i, f$ . This inhabitant is exactly the extract of  $\lambda A, B, i, f. f i$ . Theorems stating the three monad laws can be regenerated and shown with the extracts in place instead of the definitions for `return_ID` and

`bind_ID`. Extensionality can be shown if needed. Then we restate the theorem saying that these components form a monad. But instead of the definitions we use the extracts of the specification theorems.

**Theorem 11 Extracted ID is a Monad**

```
⟨ID,
  extract_of(ID_spec_bind),
  extract_of(ID_spec_return),
  extract_of(Associativity_of_ID),
  extract_of(Right Identity_of_ID),
  extract_of(Left Identity_of_ID)⟩ ∈ Monad(ℍ)
```

The proof of this theorem is identical to the first version with the exception that we must apply a conversion to the extract, unfolding it in place. Once this is completed we rerun the previous proof. This gives us a monad made up, with the exception of the type constructor, of extracts from specifications describing the behavior of the monad.

**4.4 List monad**

The list monad gives a slightly more interesting example. However specification of list operations is slightly more involved than for ID. We reduce our problem of constructing an *ListM* to the problem of constructing a list. Hence we use the systems computational notion of list operations and use the *ListM* as a wrapper to define the monad. We assume that the computational environment knows how to handle, build, such functions as concatenation and other elements of list construction. (As stated above lists are a built in type in Nuprl so by specifying a theorem about lists we have reduced this notion to a core computation in our environment.)

The theorem concerning `return_ListM` is an existential. It asserts the existence of a list, *l*, that for any *x*, *l* has *x* as a member and has length equal to one.

**Theorem 12 ListM\_return\_bind**

$$\forall A : \mathbb{U}. \\ \forall x : A. \exists L : ListM\ A. \\ x \in L \wedge (|L| = 1 \in \mathbb{N})$$

We note that the  $x \in L$  is the definition of list membership and not a typing obligation. To prove this we provide a witness *cons* (*x*, []). We then show we can find *x* in the list and a method to compute its length. We state the theorem in terms of Lists instead of ListM for clarity to the reader. The theorem in the ListM case is identical except for unfolding and

reducing the type constructor *ListM A*. The set type is used to define an object where the membership in that set is based on reasoning about indexes which characterize the behavior. We do this for pragmatic reasons in terms of proving the theorem. The result though is that the set type eliminates the evidence and provides solely the witness to the theorem. Or in this case the function that generates the list *l'* from a list *l* and a function *f*.

**Theorem 13 ListM\_spec\_bind**

$$\forall A : \mathbb{U}. \\ \forall l : ListM\ A. \\ \forall f : A \rightarrow ListM\ B \\ \{l' : ListM\ B | \\ \forall i : \{0..|l'|-\} \\ \exists j : \{0..|l|-\} \\ \exists k : \{0..|f(l[j])|\} \\ l'[i] = (f(l[j]))[k] \in B\}$$

We use list induction on *l* to prove this fact. This extract has the same computational content of *concatMap* from above. We can once again prove that the two are extensionally equal in the type of *ListM B*.

Insert new extract

Unfortunately the extract is often rather ugly with it's choice of variable names and the fact that it leaves obvious redexes unreduced. It can be cleaned up and given proper, readable, variable names.

Insert new clean extract

With these two pieces, we duplicate the procedure we laid out in the ID monad and prove that `ListM_extract_monad` is in fact an instance of the monad type similar to the *ID* case.

**5 Applications**

As stated in the introduction, our goal is to close the gap between practical programming while using constructive type theory as a functional programming language. The impetus is on us to show that this can be done. To further make our case we present several practical implementations using our monads built from extracts. The first is a reimplementaion of the *sumTree* algorithm from above. The second is a formulation of list comprehensions. We report these with the notion that they are representative of most algorithms using monads although simplistic. The final set of example applications is an exploration of

extracting monadic programs directly from the specification. The goal is to create a translation from the theorem provers term language into a more practical functional language, such as a Haskell implementation.

## 5.1 General methodologies

We are given, at least, two choices when it comes to writing applications with our formalized monads. First we can develop programs using the extracted versions or the defined versions of `return` and `bind` for each monad. This would in essence be the Haskell equivalent of declaring an instance of the monad class within a module (or application) and using it specifically as such. In the Nuprl definitions the types are part of the computation as well as being a constraint of the function type. Given the fact that we do have to explicitly declare and carry our types throughout the computation we do not have the luxury of a implicit overloading of the operators such as `bind` and `return`. We instead will have to use *bind\_ListM* and *return\_ListM* explicitly.

We can emulate, partially, the class instance definitions in Haskell by meta-programming. Nuprl's tactic language is a variant of ML which allows us to create and manipulate terms in the object language. If a new monad instance is needed which has not yet been defined and accepted as an object in our library, we can automatically construct much of the information we need to declare this monad. We have a set of meta-programs that will allow us to generate definitions and theorems equivalent to the ones given above for *ID*, *ListM*, *Maybe*, and *State*. The arguments to the meta-programs are the type and the definitions of the `bind` and `return` operator. Obviously this must be a case where we are not specifying but verifying the behavior of these operators. The meta-programs themselves generate all of the objects necessary for the acceptance of our new monad into our library of monads.

The same meta-language gives the ability to write and apply tactics that will often get close to a complete proof of the three laws as well as start to prove the theorem stating that the newly declared instance is in fact an instance of the type monad. In general the tactics are unable to complete the proof. There is often reasoning about the types themselves that is unique and hence we are obligated to complete the proof of, for instance *left\_identity\_of\_State*, by hand. However, in some cases, such as *ID*, the simple act of unfolding all of the definitions and reducing the resulting terms results in complete proofs of the

each law and the *ID\_Monad* theorem. This process has been applied with success to generating recursive types and proving their well-formedness goals. In addition the same meta-programming environment has been used to define versions of membership functions and theorems giving induction principles for the specified recursive types. Isabelle/HOL has similar facilities to build and reason about inductive types.

In general, if the definitions of the operators are known and the monadic type constructor has been defined this is the quickest way to generate a new instance of the monad. Defining each operator by hand, proving it well-formed, and then proving the laws and that it is a monad can be quite time consuming. Tactics can be written that will facilitate the proof discovery but these are again generally type specific. They are truly only useful for proving properties about that specific instance of the type (or the monad).

This level and type of reasoning often is exactly what is needed when verifying programs and when giving additional definitions and functions in terms of the monad components. For instance *sumTree* above was easily shown to be a well formed definition in the type  $ID \mathbb{N}$  using tactics designed to unfold and reduce the definitions of *bind\_ID* and *return\_ID* as well as *ID* itself. Similarly for programs written with *Maybe* and *State*. The *sumTree* and list comprehension examples below are indicative of this process of using the extracted functions directly in the resulting application.

The second choice that is available is to program solely by generating extracts through specification. This, as we have seen, can be a process of specifying the behavior of the monad instance in terms of the behaviors of `bind` and `return` then using the resulting extracts from these operators to define the monad. The program can also be derived, or extracted, directly from a specification. The functions and their specifications can become quite complex. However there appears to be a general methodology of generating and proving specifications in the less complex cases. These specifications are variants on the specification of the `bind` and `return` operator. Hence the extract incorporates this behavior directly. This will be described by example in the section on list operations. The proofs in all of the cases below are similar to the proof of the original specification. The proof scripts can be edited to provide information regarding the types and the functions used in the new specification. The resulting extract provides a function with no explicit mention of a monad but the behavior of one.

## 5.2 sumTree

We define the *sumTree* algorithm in terms of the extracts of the specifications instead of the definition of the `return_ID` and `bind_ID`.

```

sumTree2 (t) =
(letrec s(t) =
  case t:
    empty → ({ext}ID_spec_return 0)
    node(x, t1, t2) →
      {ext}ID_spec_bind (N N)
      {ext}(return_ID x )
      ({ext}ID_spec_bind (N N)
        λ y. (s t1)
      ({ext}ID_spec_bind (N N)
        λ n. (s t2)
      (λ m. {ext}ID_spec_return
        (y + n + m)))) ) t

```

Readability becomes a problem as we lose our nice infix notation but the computational results remain the same.

## 5.3 List comprehensions

List comprehensions have long been a standard application of monads. They allow functions on lists to be defined quickly and concisely. Here we define a limited version of list comprehensions which take a generator over the natural numbers and a guard.

### Definition 6 (List Comprehensions)

$$\begin{aligned}
&\forall B : \mathbb{U}. \forall n, m : \mathbb{N}. \\
&\forall P : \mathbb{P}. \\
&[f : \mathbb{N} \rightarrow (ListM\ B) | gen(n, m), P(x)] = \\
&\{ext\}ListM\_spec\_bind \\
&(\lambda x. if (P(x)) then f x else []) (gen(n, m))
\end{aligned}$$

The look of the *ListM* comprehension statement is defined by the display form. It is quite easy to prove that this term has the type *ListM B* when  $f : \mathbb{N} \rightarrow (ListM\ B)$ . *Gen(n, m)* has the type  $\mathbb{N} \times \mathbb{N} \rightarrow ListM(\mathbb{N})$ . We can prove this with either version of *ListM\_bind* although as one might expect the *concatMap* version is slightly easier to state and to manipulate. The well-formedness proofs are nearly identical.

## 5.4 List Operations

In this last section we look at specifications of functions using monadic behavior on lists. These functions which embody some rather trivial but infinitely

useful routines on lists can be defined in terms of a monad because they are easy to understand and their behavior is well understood. There is nothing unique about the specifications themselves. They are variants on the original propositions, in this case the proposition which gave us `concatMap` or *bind\_ListM*.

We specify `flatten` with a theorem that is nearly identical to *bind\_ListM*. The primary difference is that instead of stating  $\forall f : A \rightarrow ListM\ B$  we actually give implementation of  $f$  that we know to be correct. In this instance  $f$  is exactly the identity function  $\lambda x.x$ . In essence we are specifying *concatMap*  $(\lambda x.x) l$  for any  $l$  that is a list of lists of any type.

### Theorem 14 flatten\_spec

$$\begin{aligned}
&\forall A : \mathbb{U}. \\
&\forall l : ListM (ListM\ A). \\
&\{l' : ListM\ A | \\
&\quad \forall i : \{0..|l| - 1\} \\
&\quad \exists j : \{0..|l[i]| - 1\} \\
&\quad \exists k : \{0..|l[j]| - 1\} \\
&\quad l'[i] = (l[j])[k] \in A\}
\end{aligned}$$

The proof is almost identical to the proof of `ListM_spec_bind`. The types are changed slightly to give a particular instance for “A”, namely it is known to be *ListM A*. And the instance of  $f$  is removed. It is possible, and likely, and just as intuitive to state the last line of the theorem as:

$$l'[i] = ((\lambda x.x)l[j])[k] \in A.$$

The redex here and in the witness in the inductive case are discharged by a simple reduction. This reduction is not computationally evident in the resulting extract. The extract, cleaned up, is as follows.

INSERT EXTRACT

The fact that the two specification propositions and proofs were so similar lead us to believe that this correspondence is a general methodology for specifying monadic programs. In fact any list operation that will work in the context of *concatMap* can be specified in this context. In other words, if there is a function  $f$  we can quickly extract a program from a proof of the theorem that incorporates  $f$ . What is unclear is how this method will scale to work on different monads and how to specify more complex behaviors.

## 6 Conclusions and Future Work

Connecting the abstract notions generated by a theorem proving environment and practical programming has long been a goal. If for nothing else than to give justification for the formalization. We approached the idea that we could take monads, which began in the realm of theory, full circle. We recapitulate this cycle by formalizing them as programming elements and use the formal objects in a practical manner. Our exploration began not as an attempt to formalize everything that a monad encompasses. It began as an attempt to formalize a notion of state and relating constructive type theory's ability to generate useful evidence for validity. This evidence is lost in typical functional programs, or at least overlooked. Hence for us monads provide an opportunity to explicitly gather the evidence that is generated by a theorem or a computational object, to be able to define a global context in our proofs-as-programs system. The work of collecting evidence of membership as index types is ongoing and the formalization of a state monad plays a crucial role in this. We are formalizing a dependently typed predicated membership function which we first mentioned in [8]. Our claim is that there is value in this evidence beyond just using it as a claim for membership or validity. One piece of future work that we are remiss in presenting here is a program that implicitly contains a monad. So far without presenting the monad as a witness we have not been able to do this beyond trivial instances.

Our formalization of monads does indeed allow us to structure our programs in such a way that we are able to better obtain a flexibility and modularity. But more important they do provide a way, that was otherwise unavailable to us, to use a notion of state and side-effects, to provide imperative style to an otherwise functional environment. We see no road blocks to continuing to build a catalog of formalized monads along with a library of programs built on top of them. We further desire to formalize much of the core of the great body of monad literature such as the work in Luth, Ghani, Moggi, Wadler, others [13], [20], [15], [19] to name only a few. As there are other constructive type theories, these explorations are not limited necessarily to Nuprl. To better serve a growing functional programming community we are constructing a system to transform programs in Nuprl's term language to a more practical language such as Haskell.

ADD????

We do not yet have a transformation from Nuprl's

term language into Haskell. There is work under way to build an interpreter from the term language into Haskell or modify Nuprl's extraction machinery so that it generates Haskell code instead of or in addition to the term language. (We are currently emphasizing the former.) We note that Nuprl's type system is richer than Haskell's, but the computation language is equivalent.

## References

- [1] All about monads version 1.1.0.
- [2] Monad prompt package 1.0.0. <http://www.haskell.org/cgi-bin/hackage-scripts/package/MonadPrompt-1.0.0.1>.
- [3] The haskell 98 language report, 2002.
- [4] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying haskell programs using constructive type theory. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 62–73, New York, NY, USA, 2005. ACM.
- [5] Lennart Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.
- [6] Richard Bird. *Introduction to Functional Programming*. Pearson Education, 1998.
- [7] James Caldwell and Josef Pohl. Constructive membership predicates as index types. In *Proceedings of PLPV '06: Programming Languages Meets Program Verification*. ENTCS, 2006.
- [8] James Caldwell and Josef Pohl. Constructive membership predicates as index types. *Electronic Notes in Theoretical Computer Science*, 174(7):3–16, 2007.
- [9] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986. Up to date information available at <http://www.nuprl.org>.
- [10] Jeremy E. Dawson. Compound monads in specification languages. In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 3–10, New York, NY, USA, 2007. ACM.

- [11] Chuan kai Lin. Programming monads operationally with unimo. *SIGPLAN Not.*, 41(9):274–285, 2006.
- [12] David J. King and Philip Wadler. Combining monads. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 134–143, London, UK, 1993. Springer-Verlag.
- [13] Christoph Lüth and Neil Ghani. Composing monads using coproducts. *SIGPLAN Not.*, 37(9):133–144, 2002.
- [14] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
- [15] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS’89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.
- [16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [17] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
- [18] Tim Sheard. Putting Curry-Howard to work. In *Haskell ’05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, New York, NY, USA, 2005. ACM Press.
- [19] P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.
- [20] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.