Structuring functional programs by using monads

Davor Obradovic, University of Pennsylvania

May 1998

Abstract

The notion of monads originates from the category theory. It became popular in the programming languages community after Moggi proposed a way to use monads to structure denotational semantics. Wadler and others showed how this can be effectively used as a methodology for building interpreters.

Monads are capable of capturing individual language features in a modular way. This paper evaluates two modular approaches for building monadic interpreters - Steele's *pseudomonads* and Liang, Hudak & Jones's *monad transformers*. We also consider the the problem of feature interaction in the monad transformers setting.

1 Monads: Introduction and Motivation

Is there a practical use for category theory? Certainly - the programming language theory is a good example. Many categorical concepts led to important discoveries about programming languages and vice-versa, many ideas from the category theory were motivated by the programming languages research. Monads probably fit in both groups.

Category theorists invented monads in the 1960's to express certain properties of universal algebra. Two decades later people started studying programming language aspects of monads. Among the biggest contributors to this promising idea were Eugenio Moggi [6] and Philip Wadler [7]. The original idea was to use monads for presenting the denotational semantics of complex languages. Monads seemed to be able to capture a variety of commonly found language properties. At the same time they can be used as a clean and extensible technique for implementing these features in interpreters. The key feature is that monads provide us with a convenient notion of a *computation*. Using this, we can enrich our computational model by distinguishing between the values and the computations. For example, two programs (computations) that calculate the same answer, but generate different screen outputs should certainly be considered different. This is hard to achieve in a setting where computations are viewed statically, as pure functions. Using monads, we can precisely specify the desired level of distinction between computations.

This balances the tradeoff between impure and pure functional languages from the language implementor's point of view. On one hand, pure languages, such as Haskell, benefit from the power of equational reasonong. On the other hand, many desired features seemed to be very hard to implement without using impure constructs, such as arrays, references and certainly I/O. Monads provided solutions that combined the best of both approaches. We should note that monads are not special programming language constructs - they are simply an example of a good data abstraction. A particularly good data abstraction! Hence, we should rather say that a program is written in monadic style, than that it uses monads. This is precisely one of the big advantages of monads; apart from the support for higher order functions, they don't impose almost any restrictions to the underlying programming language environment. Monadic style is just a simple methodology that turned out to be surprisingly general in a variety of situations.

It can capture individual properties while keeping the abstraction level appropriately high, thus saving us from too much concern about technical details.

1.1 What is a monad?

As mentioned earlier, the concept of monads was coined in the category theory and later adopted by computer scientists. This section describes monads from the functional programming and categorical standpoint. We will use the standard Haskell notation (with some additions in the chapter 4) throughout the paper. The reason is that Haskell, as a purely functional programming language, is very close to the actual mathematical language commonly used by the semanticists. This makes the formal reasonong about our interpreters easier.

A functional programmer thinks of a monad as a triple (M, unitM, bindM), where M is a type constructor, unitM and bindM are polymorphic functions

unitM :: a -> M a bindM :: M a -> (a -> M b) -> M b,

and the following laws are satisfied:

• Left unit:

(unitM a) 'bindM' k = k a

• Right unit:

```
m 'bindM' unitM = m
```

• Associativity:

```
m 'bindM' (\a->((k a) 'bindM' h)) =
= (m 'bindM' (\a-> k a)) 'bindM' h
```

for every

```
m :: M a
k :: a -> M b
h :: b -> M c.
```

Notice the Haskell syntax for the infix version of bindM which is 'bindM' and for a lambda abstraction $\lambda x.e$ which is written as $x \rightarrow e$.

Intuitively, we can think of a type M a as the type of computations resulting in a value of type a. unitM embeds values into computations, such that if v::a is a value, then unit v is the computation that does nothing except yields the value v. Similarly, if m::M a is a computation and $k::a \rightarrow M$ b is a function, then m 'bindM' k is a computation that performs m, applies k to the resulting value and *then* performs the computation returned by k. The word "*then*" is very important here, since 'bindM' will indeed be used to control the order of evaluation.

For a given monad M, we can also define the following polymorfic functions:

joinM :: M (M a) -> M a mapM :: (a -> b) -> (M a -> M b)

with

joinM m = m 'bindM' id
mapM f m = m 'bindM' (\a -> unitM (f a)).

It is easy to see that with the appropriate types the following holds in every monad M:

```
    (1) joinM . joinM = joinM . (mapM joinM)
    (2) joinM . unitM = id
    (3) joinM . (mapM unitM) = id.
```

The symbol '.' is here used to denote the functional composition.

As far as category theorists are concerned, a monad over a category C is a triple (T, u, j), where $T : C \to C$ is a functor, $u : Id_{\mathcal{C}} \to T$ and $j : T^2 \to T$ are natural transformations and the following diagrams commute:

The two definitions are (as expected) isomorphic in a certain sense. We can regard C as the category whose objects are our types and arrows are functions with standard functional composition and identity. The functor T acts on objects (types) as the type constructor M and on arrows (functions) as mapM. Polymorphic functions unitM and joinM correspond to the natural transformations u and j respectively. Finally, the first commutative diagram corresponds to the derived rule (1), while the second one corresponds to (2) and (3).

1.2 Some simple monads

Let's see a couple of simple examples. The simplest one is the identity monad I:

```
type I a = a
unitI a = a
a 'bindI' k = k a
```

The type constructor I is the identity and so is the function unitI. bindI is just the ordinary functional application. This monad, as the extreme case, exactly identifies values with computations.

A more interesting example where this is not the case is the list monad L. Haskell syntax for the type of lists whose elements are of the type **a** is [**a**]. We also assume the existance of the standard *list* functions map :: $(a \rightarrow b) \rightarrow ([a] \rightarrow [b])$ that maps a given function over a list and join :: $[[a]] \rightarrow [a]$ that "flattens" a list of lists into a single list. The list monad is given by

```
type L a = [a]
unitL x = [x]
l 'bindL' f = join (map f l).
```

Notice that the *monadic* functions mapL and joinL are really the standard map and join *list* functions, just as we would normally expect.

Another similar example is the tree monad:

```
type Tree a = Leaf a
| Node (Tree a) (Tree a)
```

Type Tree a is the type of regular binary trees whose leaves hold values of type a. bind-ing the function f to a tree results in a new tree obtained from the original one by "appending" to each leaf a new tree. This new tree is obtained by applying f to the value stored in the leaf.

2 Monadic Interpreters

We are now ready for some more interesting examples. In this section we will see how monads can be used to implement a simple interpreter for lambda calculus, as suggested by Philip Wadler in [8]. Wadler demonstrated the flexibility of this idea by showing how to build a couple of quick variations of the interpreter.

The interpreter deals with terms and values. Terms consist of variables, integer constants, additions, lambda abstractions and applications. A value is either Wrong (indicating an error), a number, or a function. When implementing an interpreter, the key idea is to interpret a function of type $a \rightarrow b$ in our target language as a function of type $a \rightarrow M a$ in our meta-language (Haskell). Nice thing is that we can do that independetly of the actual monad M. Description of M, which includes details about how are computations exactly represented and carried out, is completely separated.

Given any M, we can think of unitM as the *identity* function and bindM as the *application* in that particular monad. The whole communication between the monad and the interpreter is done through those two operations. Here is the actual code for the interpreter:

```
type Name = String
data Term = Var Name
| Con Int
```

```
Add Term Term
          Lam Name Term
          App Term Term
data Value = Wrong
           Num Int
           | Fun (Value -> M Value)
type Environment = [(Name, Value)]
showval :: Value -> String
showval Wrong = "<wrong>"
showval (Num i) = showint i
showval (Fun f) = "<function>"
interp :: Term -> Environment -> M Value
interp (Var x) e = lookup x e
interp (Con i) e = unitM (Num i)
interm (Add u v) e =
  interp u e 'bindM' (\a ->
  interp v e 'bindM' (b \rightarrow add a b))
interp (Lam x v) e =
  unitM (Fun (a \rightarrow interp v ((x,a):e)))
interp (App t u) e =
  interp t e 'bindM' (\f ->
  interp u e 'bindM' (\a -> apply f a))
lookup :: Name -> Environment -> M Value
lookup x []
                  = unitM Wrong
lookup x ((y,b):e) = if x==y
                     then unitM b
                     else lookup x e
add :: Value -> Value -> M Value
add (Num i) (Num j) = unitM (Num (i+j))
add a b
                    = unitM Wrong
apply :: Value -> Value -> M Value
apply (Fun k) a = k a
                = unitM Wrong
apply f a
test :: Term -> String
test t = showM (interp t [])
```

Notice that in the rule for application both the function and the argument are evaluated, so this is a call-by-value interpreter. As we'll shortly see, a small modification would implement the call-by-name strategy. But let's first give some meaning to the above interpreter.

2.1 Ordinary interpreter

Substitute the identity monad I from 1.2 for M in the above interpreter while defining

```
showI = showval.
```

If we try to test it by evaluating test term0, where

conveniently written as $term_0 = (\lambda x.x + x)(10 + 11)$, we get 42 as the result, as expected.

2.2 Error messages

Error reporting can be added to the interpreter by defining the following monad:

```
data E a = Suc a | Err String
unitE a = Suc a
(Suc a) 'bindE' k = k a
(Err s) 'bindE' k = Err s
srrorE s = Err s
showE (Suc a) = "Success: " ++ showval a
showE (Err s) = "Error: " ++ s
```

As before, we have to replace the monad M by E in the interpreter. Notice how an error, as soon as discovered, percolates to the top level via bindE function. To introduce error messages, we simply replace every occurence of unitE Wrong by the appropriate call to errorE:

```
lookup x [] =
  errorE ("unbound variable: " ++ x)
add a b =
  errorE ("should be numbers: " ++
      showval a ++ "," ++ showval b)
```

```
apply f a =
  errorE ("should be a function: " ++
      showval f)
```

Evaluating test (App (Con 1) (Con 2)) returns "Error: should be function: 1".

2.3 Output

We can modify our interpreter to include the output features. The following monad, when included in the interpreter, accumulates the output during the evaluation in the order it occurs.

We pair each value with the output generated for that value. unitO simply returns a value without any output, while bindO does the application and concatenates the generated outputs. Notice how 'bindO' enforces the correct order of generating output strings. r will be generally evaluated before s, because of the dependency introduced by a.

This just describes how the output is propagated. To generate the output, we introduce a new function out0. We also need to extend the language of terms and the interpretation function:

```
out0 :: Value -> 0 ()
out0 a = (showval a ++ "; ", ())
data Term = ... | Out Term
interp (Out u) e =
    interp u e 'bind0' (\a ->
    out0 a 'bind0' (\() -> unit0 a))
```

output buffer at the same time. An interesting detail to notice is that **bind0** indeed behaves similarly to the sequencing operator in the imperative languages (usually denoted by a semicolon). For the above function body, one would write the following similar imperative code:

```
a := interp (u,e);
out (a);
return (a);
```

bind will typically behave like a sequencer in monads with states.

2.4 Nondeterminism

If we want to model nondeterminism, the results are going to be lists of values, rather than single values. The way to do that is, of course, to use the list monad L from 1.2 together with

```
showL m = showlist (map showval m).
```

The function showlist translates a list of strings into a single string. Now we include this monad in the interpreter together with the following updates:

```
zeroL = []
l 'plusL' m = l ++ m
data Term = ... | Fail | Amb Term Term
interp Fail e = zeroL
interp (Amb u v) e =
    interp u e 'plusL' interp v e
```

Terms can now be "ambiguous", so (Amb u v) results in the list of all the values that are results of u or v. For example, evaluating

Interpreting (Out u) returns the value of u as the result and sends it off to the output, emptying the yields "[4,6]".

2.5 Call-by-name

When using the call-by-value evaluation strategy, functions are applied only to *evaluated* arguments, so the type of functional values is Fun (Value -> M Value). In the call-by-name interpreter, arguments are passed unevaluated to functions, so functional values will be represented with the type Fun (M Value -> M Value). All the changes are shown below. The apply function basically looks the same, but it has a different type.

```
apply :: Value -> M Value -> M Value
```

Variables will now be bound to "computations", rather than their final values, hence the change in the environment type. The lookup function now does not have to "wrap" its result in monadic type anymore, since the result is already "wrapped". Finally, the change in the application rule reflects the fact that the argument u is passed unevaluated to the evaluated function.

We can combine call-by-name evaluation with all the previously described enhancements. For example, nondeterministic version of the call-by-name interpreter would evaluate the old example

test

```
(App (Lam "x" (Add (Var"x")(Var"x")))
(Amb (Con 2)(Con 3)))
```

to "[4,5,5,6]", rahter than "[4,6]" what we had before.

2.6 Overview

Following the same general idea, we can easily extend our interpreter to deal with continuations, references, call-by-need evaluation and beyond. This sounds almost too good to be true. Let's briefly summarize the main points and problems in Wadler's approach:

- This approach benefits from a reasonably good separation between the interpreter and the underlying monad.
- Uniformity of the monadic approach makes the implementation clean, simple and easy to understand.
- The *type* information becomes more valuable. Some vital properties are visible just by looking at the types (recall the call-by-name modification and the difference in types of functional values). Now, for instance, we can easily localize the parts of a program that have side-effects from those that don't. Usage of different stateful monads in Haskell is more closely studied in [2] and [4].
- The problem of interaction between multiple monadic features in a single interpreter is not addressed - how does one gradually build a monad that captures multiple features?
- Finally, efficiency of a monadic implementation might be a drawback. Monadic implementations of many common features may require heavy use of higher-order functions (references and continuations being typical examples). Certain common monads hopefully could be optimized and perhaps even be provided as language primitives.

3 Composing Monads

Guy Steele really liked Wadler's work! But some things were not good enough. First, Steele wanted to be able to incrementally add new features to the existing interpreter by precisely specifying how they work together. In [3] he showed how to construct monadic *building blocks* that one can stack together (almost like Legos) to obtain monads. In the previous chapter we saw that Wadler's interpreter, although flexible, usually requires some modifications of data structures to be done *by hand* for each new feature. Steele proposed a way to make this process more automatic.

The main idea was to use *pseudomonads*. A pseudomonad behaves like a monad with a "hole". When a monad is composed with a pseudomonad in a certain sound way, the result is a new monad. It can then be repeatedly composed with more pseudomonads, thus enriching the interpreter. The interpreter again deals with *terms* and *values*. They now exist on multiple levels, since we have multiple levels of monads. Terms and values are hierarchically connected with projections and form *towers of data types*. The next chapter describes monads and pseudomonads in Steele's environment.

3.1 Monads and pseudomonads

Guy Steele introduces "monads" and pseudomonads through the "Haskell" code that supports them (the reason for quotation will be clear shortly). Notice that names Monad and Pseudomonad are used for both type constructors and data constructors.

```
type Unitfn p q = p -> q
type Bindfn p q = q -> (p -> q) -> q
type Pseudobindfn p q =
Monad q r -> q -> (p -> r) -> r
data Monad p q =
Monad (Unitfn p q) (Bindfn p q)
data Pseudomonad p q =
Pseudomonad (Unitfn p q) (Pseudobinfn p q)
unit :: Monad p q -> Unitfn p q
unit (Monad u b) = u
bind :: Monad p q -> Bindfn p q
bind (Monad u b) = b
pseudounit :: Pseudomonad p q ->
```

Unitfn p q pseudounit (Pseudomonad u pb) = u pseudobind :: Pseudomonad p q -> Pseudobindfn p q pseudobind (Pseudomonad u pb) = pb

Wadler defines a monad in Haskell as a triple (M, unitM, bindM) containing one type constructor and two polymorphic functions. Steele, on the other hand, defines monads as 4-tuples (p,q,u,b) containing two types and two (not necessarily polymorphic) functions. Intuitively, Steele would like to treat type q as type M p. But Wadler's func $bindM :: Ma \rightarrow (a \rightarrow Mb) \rightarrow Mb$ tion isstillstrictly more polymorphic than $b :: q \rightarrow (p \rightarrow q) \rightarrow q$. These two definitions neither match exactly, nor there is an isomorphism that would establish a natural 1-to-1 correspondence between the two abstract data types. Each monad (M, unitM, bindM) corresponds to (but is not completely described by) a family of Steele's monads of the form (p, M p, u, b). Wadler's monads are polymorphic, since they deal with type constructors, while Steele's monads appear to be their special instances.

For example, we can define identity monads of the type Monad p p for each type p with

idmonad :: Monad p p idmonad = Monad $(\langle x \rightarrow x \rangle) (\langle z \rangle k \rightarrow k z)$

Pseudomonads essentially are monads parametrized by monads. A pseudomonad encapsulates two types (p and q) and operations pseudounit and pseudobind. pseudounit has the role of a unit function for ordinary monads, while pseudobind is a kind of bind parametrized by a monad. It is convenient to keep the infix notation by defining operators <<, # and >> in Haskell in such a way that we can write x <<m>> f for bind m x f and x <<m#p>> f for pseudobind p m x f. Pseudomonads, just like monads, are required to have certain natural properties:

• Left unit:

unit p a <<m p>> f = f a

• Right unit:

```
x \ll p >> (h . unit p) = h x
```

• Associativity:

```
x <<m#p>> (\a-> f a <<m#p>> g) =
(x <<idmonad#p>> \a-> f a)<<m#p>> g
```

Now we have to specify how do we actually compose monads with pseudomonads. It is done by using the operator &.

```
m&p = Monad
  (unit m . pseudounit m)
  (\z k -> z <<m>>> (\w -> w <<m#p>> k))
```

If m is a monad and p is a pseudomonad, then m&p might be a new monad. Left and right unit monad laws are guaranteed to hold, but associativity needs to be checked separately. Monads generally do not compose!

But there is another, perhaps more serious problem in the given Haskell code. The definition of the Pseudobind type has a free type variable r and hence is not legal in the language! Steele's motivation for such declaration comes from the fact that we don't a-priori have the complete type information about monads that we eventually want to compose with a given pseudomonad. Ideally, we would like to treat the type variable r existentially. That is a reasonable requirement, although not quite compatible with the current Haskell type system. Probably the easiest way around this problem is to include a type variable \mathbf{r} in the declaration as type Pseudobindfn p q r = ..., but Steele rejects this as a cumbersome solution. He instead uses a preprocessor called the program simplifier which circumvents certain language restrictions and produces a correct Haskell program.

As expected, the type of the composition operator is

& :: Monad q r -> Pseudomonad p q -> Monad p r.

It is important not to confuse the operators # and &. Expressions m#p and m&p do not bear any special resemblance apart from the fact that # is used in the definition of &. In fact, they even have the different types. For $f :: p \rightarrow r$, the expression x <<m#p>> f makes sense for x :: q, whereas x <<m&p>> f requires x :: r.

As one would normally expect, we can compose pseudomonads in a natural way, thus obtaining more pseudomonads. If pa and pb are pseudomonads, then their composition pb&pa satisfies

m&(pb&pa) = (m&pb)&pa.

3.2 Building an interpreter

An interpreter is a mapping from *terms* to *values*. When adding a new feature, both of those types generally need to be modified. Following that idea, Guy Steele introduces the notion of *building blocks*. A building block is a function that takes an interpreter from terms of type t to values of type v and produces a new interpreter that maps terms of type t' to values of type v'. It does so by using two lifting pseudomonads, one of the type Pseudomonad t t' and the other of the type Pseudomonad v v'.

Multiple composition of building blocks generates two "towers" of data types - one for the terms and one for the values. Complete interpreter on each level is given by the corresponding *package*. A package is a set (or rather a list) of routines and/or data that implement certain *fixed* set of vital functions for the interpreter. In our case, a package can contain routines of the following eight kinds:

- ParseR parsing
- InterpR interpreting
- ShowvalR printing
- ComplainR error reporting
- MakenumR constructing number values
- MakefunR constructing function values

• ApplyR - applying functions

• NameR - a string that names the interpreter

We immediately see inflexibility of such approach. The above set of routines is not nearly exhaustive for the kinds of interpreters we would like to consider. That is certainly a disadvantage of this approach. Or rather, that is the price we have to pay for making things a bit more automatic. We can certainly benefit from the fact that we don't have to change the data structures by hand, but at the same time we are limited to constructing only interpreters of certain kinds.

Another "meta-objection" is about treating a parser as a fundamental component of a package. Parser is completely irrelevant to the monadic implementation of the interpreter. Steele should have probably been less concerned with technical issues such as parsing, since it makes some parts of Haskell code in his paper [3] very tedious to read.

Each package carries the type information about the *current* types of terms and values (t and v), *top* types of terms and values (t'' and v'') and types of values stored in the environment (ve). Below are the datatype definitions a package.

```
data Routine t v t'' v'' ve =
    ParseR (String -> [(t'', String)])
    InterpR (t -> [(String, ve)] -> v'')
    ShowvalR (v -> String)
    ComplainR (String -> v'')
    MakenumR (Int -> v'')
    MakefunR ((v'' -> v'') -> ve)
    ApplyR (v'' -> v'' -> v'')
    Name String
data Package t v t'' v'' ve =
```

```
Package [Routine t v t'' v'' ve]
```

Notice that the interpreter accepts a term of the current type, but produces a value of the top type. This is actually a value of the current type projected to the top of the value type tower via the unit func-

Т	//	inte <u>rp</u>	$\stackrel{r \text{ top}}{\rightarrow}$	V''
1 1	· ·			↑ ↑
Т	2	interp	$\stackrel{\text{or } P_2}{\rightarrow}$	V_2
1				1
Т	1	interp	$\stackrel{\text{or } P_1}{\rightarrow}$	V_1
1	ŀ			1
Т	0	interp	$\stackrel{\text{or }}{\rightarrow} P_0$	V_0
mT_2 mT_1	::	Pseuc Pseuc	iomo: lomo:	nad $T_1 T_2$ nad $T_0 T_1$
mV_2 mV_1	::	Pseud Pseud	iomo: lomo:	nad $V_1 V_2$ nad $V_0 V_1$
P_2 :: P_1 :: P_0 ::	Pa Pa Pa	ckage : ckage : ckage :	$\vdots T_2 \ V_2 T_1 \ V_1 T_0 \ V_0$	$T'' V'' V_E$ $T'' V'' V_E$ $T'' V'' V_E$

We also assume the existence of functions interpr, parser, complainr, etc. that when given a package, extract the corresponding component from it.

3.3 Base of the tower

Here we describe how to construct a simple base interpreter on which all the other type towers can be erected. The only term it interprets is Bogon (projected to the top of the term tower) and the only resulting value is Wrong (projected to the top of the value tower).

data TermZ = Bogon data ValueZ = Wrong

tion.

```
interpreter tmt tmv top = Package
 [ParseR parseZ, InterpR interpZ,
 ShowvalR showvalZ, ComplainR complainZ,
 NameR nameZ]
where
 parseZ s = [(unit tmt Bogon, s)]
 interpZ Bogon _ =
    complainr top "invalid expression"
    compainZ s = unit tmv Wrong
    showvalZ Wrong = "<wrong>"
    nameZ = "interpreter"
```

The function interpreter takes three parameters and produces a package. Such function is called a *prepackege*. The parameters are top monad for terms tmt, top monad for values tmv and the top package top. tmt and tmv are used for projecting terms and values to the top of the towers. top is here used for reporting errors, since we have to report them at the top level. To complete a prepackage, we need to give it some suitable parameters. Here we are building the base level, so top and bottom levels are the same. Therefore, the identity monad will a be good candidate for both tmt and tmv. Where do we get the top package from? Well, we are just working on constructing one. Hence, we can use a little lazy trick - define the top cyclically in terms of itself.

```
complete prepkg = top
  where top = prepkg idmonad idmonad top
type Term = TermZ
```

```
type Value = ValueZ
interp_pkg = complete interpreter
```

The complete function "ties a knot" at the top and returns the top package. This will typically be used in all further refinements of the interpreter. We will first stack all the pseudomonads (by stacking the corresponding building blocks) thus obtaining a prepackage. Then we can simply apply the complete function to finish the construction and yield the top package.

Steele seemed to have problems with recursive definitions in Haskell, such as the above one. This is another point where the program simplifier had to intervene.

3.4 The first floor

This section describes how can one extend an interpreter by adding another building block. We describe an example of the numbers building block.

The term type is extended with constant constructor and addition, while the value type is extended with numerical values.

```
data TermN t'' t =
 Con Int | Add t'' t'' | OtherTN t
data ValueN v'' v = Num Int | OtherVN v
mTN = Pseudomonad
        (x \rightarrow OtherTN x) mTNbind where
 mTNbind m (Con x) f = unit m (Con x)
 mTNbind m (Add x y) f = unit m (Add x y)
 mTNbind m (OtherTN x) f = f x
mVN = Pseudomonad
        (x \rightarrow OtherVN x) mVNbind where
 mVNbind m = qxfoo where
    qxfoo (Num x) f = unit m (Num x)
    qxfoo (OtherVN x) f = f x
numbers oldprepkg tmt tmv top =
  update oldpkg [<list of new routines>]
where
  oldpkg = oldprepkg
             (tmt & mTN) (tmv & mVN) top
 parseN ...
  interpN (Con x) _ = unit tmv (Num x)
  interpN (Add x y) env =
    interpr top x env <<tmv>> (\u ->
    interpr top y env <<tmv>> (\v ->
    case (u,v) of
      (Num j,Num k) -> unit tmv(Num(j+k))
      (\_, \_) \rightarrow complains top (
        "should be numbers: " ++
          showval top (unit tmv u) ++ ", "
       ++ showval top (unit tmv v))
    ))
```

```
interpN (OtherTN x) env =
    interpr oldprepkg x env
showvalN (Num x) = show x
showvalN (OtherVN x) = showvalr oldpkg x
makenumN x = unit tmv (Num x)
nameN = ...
```

The numbers function is an example of a building block. It takes an old prepackage and produces a new prepackage. In our case, the old prepackage is interpreter. To get the new package, we need to *complete* the resulting new prepackage. We can do it by using the function complete. It will compose the pseudomonads mTN and mVN with the identity monad, leave top to be the fixpoint (the "knot") as before and return the final package. The code for that is

```
type Term = TermN Term TermZ
type Value = ValueN Value ValueZ
interp_pkg = complete(numbers interpreter).
```

We can perform multiple extensions in a similar fashion. At every stage, the interpreter will typically be evaluated by an expression of the form

```
complete (bn(...(b2(b1 interpreter))...)),
```

where b1, b2, ..., bn are some building blocks. We see that the building blocks composition is generally not commutative, much like the functional composition. An interesting problem would be to find the optimal order of building blocks for a given set of features. More general, what kinds of issues are involved in comparing two building-blocks-based implementations of the same set of features?

3.5 The program simplifier

The program simplifier in Guy Steele's environment has a multiple function:

- Inlining of certain functional definitions
- β -reductions and α -conversions of terms in a way that keeps them simple and readable

- Substitution and simplification of the explicitly provided type declarations Term and Value
- Circumvention of some of the problems that the original type checker had with suggested declarations

Most of the transformations done by the program simplifier are sound. They are usually based on simple equational reasoning and any Haskell compiler can perform them if it finds them useful. But some of the transformations attempt to "correct" certain parts of Steele's code that failed to produce the desired effects or were even rejected as erroneous by the compiler! Introducing such transformations is not a good idea for at least two reasons:

- They are potentially unsound.
- Even if they are sound, such rules are not standard in the language, so everyone who attempts to use such tool needs to know exactly how they work. This can be an obstacle for the wide addoption of the approach.

Nevertheless, we do benefit from the program simplifier - it makes our implementation work!

3.6 Overview

Here we give an overview of the basic *pros* and *cons* of the Steele's approach for building monadic interpreters.

Pros:

- Making monads extensible through the use of pseudomonads.
- Dealing with multiple upgrades in a modular fashion.

Cons:

- Nonstandard definition of a monad, not clearly related to the "standard" definition.
- Fixed set of possible routines that a package can contain.

- Using *lists* of routines for representing packages. This can be inefficient, especially if we extend the notion of a package.
- Tight dependence on the program simplifier whose actions are not clearly defined.

4 Monad Transformers

Philip Wadler described a general monadic methodology for building programming language interpreters. His technique, although very promising, isn't modular in its nature. He builds one monad for each interpreter from the scratch. This works well in certain simple cases, but the job of creating a good monad becomes harder as the number of desired features grows. The question is how to divide this job into multiple steps? Previous chapter described Guy Steele's attempt to solve this problem by introducing pseudomonads that can be chained together to form more complex monads. This approach wasn't very general and its implementation faced various problems due to certain language restrictions of Haskell. One of the biggest problems was the type system. This chapter describes a work of Shieng Liang, Paul Hudak and Mark Jones [5] which tries to remedy Steele's problems. They used the *Gofer* language environment, which is an enhanced variant of Haskell. Their key notion is that of a monad transformer which generalizes pseudomonads. This idea is essentially simpler and more general than the Steele's idea. They also treat the problem of lifting operations from lower layers to higher layers of monad transformers in a natural way.

4.1 The Gofer environment

In this section we give a brief introduction to the Gofer constructor class system through a couple of illustrative examples. The idea behind constructor classes is similar to the idea of modules (e.g. *structures* in SML). We want to be able to generically express some common features of types or type constructors. Consider, for instance, the *map* function in Haskell

map :: (a -> b) -> [a] -> [b]

which maps a function over a *list*. We may be interested in a similar function for mapping over *trees*.

More general, we may want to do that for arbitrary number of type constructors. For that purpose, Gofer provides a way to define signatures for type and constructor *classes* and their *instances*.

```
class Functor t where
 map :: (a -> b) -> t a -> t b
instance Functor List where
 map f [] = []
 map f (x:xs) = f x : map f xs
instance Functor Tree where
 map f (Leaf x) = Leaf (f x)
 map f (Node l r) =
 Node (map f l) (map f r)
```

It is very natural to define a constructor class for monads. Functions map and join are automatically derived from unit and bind as described in 1.1.

```
class Monad m where
  unit :: a -> m a
  bind :: m a -> (a -> m b) -> m b
  map :: (a -> b) -> m a -> m b
  join :: m (m a) -> m a
  map f m = m 'bind' (\a -> unit (f a))
  join z = z 'bind' id
```

Gofer allows us to elegantly define terms and values as *extensible union types*. Steele had to use his program simplifier for that job (i.e. to "flatten" the data types from multiple levels into a single union type). This consequently eliminates most of the Steele's typing problems. Traditionally, disjoint union of types is constructed through the OR operator as shown below. data OR a b = L a | R b

L and R are injections of "subtypes" into a "supertype". Conversely, a value of the "supertype" can be projected to one of the "subtypes" using pattern matching. Generally, we can define the subtyping relation by specifying the injection and the projection.

```
class SubType sub sup where
inj :: sub -> sup
proj :: sup -> Maybe sup
data Maybe a = Just a | Nothing
instance SubType a (OR a b) where
inj = L
prj (L x) = Just x
prj _ = Nothing
```

Now we can express the subtyping relation that can reach through all the levels of OR nesting.

```
instance SubType a b =>
        SubType a (OR c b) where
inj = R . inj
prj (R a) = prj a
prj _ = Nothing
```

For example, if we declare

type Value = OR Int (OR Fun ()),

the type checker will automatically infer that Int and Fun are both subtypes of Value.

4.2 Interpreters in Gofer - basics

There are three main components to the interpreter as seen by Liang, Hudak and Jones: type Term, type Value and monad InterpM. Following the Wadler's idea, InterpM encodes computations. We call it the interpreter monad of final answers. Those three components are independent and their "richness" depends on the complexity of the interpreter we want to build. Interpreting function maps terms to computations of values.

```
interp :: Term -> InterpM Value
```

We can now define a class **InterpC** of term types that can be interpreted.

```
class InterpC t where
  interp :: t -> InterpM Value
```

Gofer can automatically build instances of that class for unions of term types.

4.3 Monad transformers

Wadler's idea about building a single monad for an interpreter becomes hard for realization when we want to implement several structurally different features, because we need to specify how each feature interacts with each other. The idea behind monad transformers is to capture features individually in an incremental way - build each next feature on the top of what we already have. This will, however, require some additional work in order to lift everything to the common top level.

A monad transformer is a type constructor t such that if m is a monad, so is t m. We want to be able to embed m - computations into t m - computations, so we need a lift operator which is a member function of the class of monad transformers.

```
class (Monad m, Monad (t m)) =>
    MonadT t m where
    lift :: m a -> t m a
```

We want this embedding to be natural in the sense that it doesn't change the nature of the existing computations. Formally, we require that every monad transformer satisfies the following two laws:

```
• lift . unitM = unitTM
```

```
• lift (m 'bindM' k) =
  (lift m) 'bindTM' (lift . k)
```

Here unitM and bindM refer to the monad m, while unitTM and bindTM refer to the monad t m.

4.4 Error monad transformer

Here we give an example of a very simple monad transformer and an interpreter based on it. The monad transformer ErrorT transforms a monad into an error monad. There are three typical steps in defining a monad transformer. First we define the type constructor:

```
data Error a = Ok a | Error String
type ErrorT m a = m (Error a)
```

Gofer system deduces that m in the above declaration is a type constructor. The next step is to define the transformation of monads through the introduced type constructor.

```
instance Monad m => Monad (ErrorT m) where
unit = unit . Ok
m 'bind' k =
m 'bind' \a ->
case a of
  (Ok x)      -> k x
  (Error msg) -> unit (Error msg)
```

The last step is to establish that ErrorT is a monad transformer by defining the lifting operator.

Since Ok :: a -> Error a, the map function of m will satisfy map Ok :: m a -> m (Error a), which shows that the defined lift has the appropriate type. It is easy to check that such lift indeed naturally embeds computations into computations with errors. If we want to play some more, we can define a special class of error monads which are capable of encoding erroneous computations via the special member function err. We also establish the fact that for every monad m, monad ErrorT m is an error monad.

```
class Monad m => ErrMonad m where
err :: String -> m a
instance Monad m =>
```

```
ErrMonad (ErrorT m) where
err = unit . Error
```

The unit function from the above definition is from the monad m.

We are now ready to construct the interpreter for a small arithmetic language, following the metodology described in 4.2. Our interpreter monad will be the "error-transformation" of the identity monad I. The final interpreter is simply an instance of the class InterpC.

```
type Term = TermA
type Value = OR Int ()
type I a = a
type InterpM a = ErrorT I a
data TermA = Num Int
           Add Term Term
instance InterpC TermA where
  interp (Num x)
                   = unitInj x
  interp (Add x y) =
    interp x 'bindPrj' \i->
    interp y 'bindPry' \j->
    unitInj ((i+j)::Int)
 unitInj = unit . inj
 m 'bindPrj' k =
    m 'bind' a \rightarrow
    case (prj a) of
      Just x -> k x
      Nothing -> err "type error"
 err :: String -> InterpM a
```

The above interpreter is somewhat artificial, since it is impossible to introduce an error. Indeed, every expression of the type **TermA** can be evaluated into a number. However, if we change the declaration of **Term** into something like

type Term = OR TermA (),

we would obviously have the possibility of errors.

4.5 Environment monad transformer

Suppose we want to extend our arithmetic interpreter to deal with functions. We need to extend our Term type and define appropriate operations that deal with functional computations. Our terms can now be functional or arithmetic expressions (including the combinations).

type Name = String

LambdaN and LambdaV perform respectively call-byname and call-by-value abstractions. We also assume the existence of the following functions for handling the environment:

```
lookupEnv :: Name -> Env ->
    Maybe (InterpM Value)
extendEnv :: (Name, InterpM Value) ->
    Env -> Env
```

How do we carry out computations in environments? Generally, we can transform every monad into an environment monad through the use of the *environment* monad transformer. If r is the type of environments and m is a monad, then

type EnvT r m a = r -> m a

is the type of m-computations in environments. The corresponding monad transformation is given below.

instance Monad m => Monad (EnvT r m) where unit a = \r -> unit a m 'bind' k = \r -> m r 'bind' \a -> k a r

The unit computation ignores the environment and returns the unit computation in m. To evaluate m 'bind' k in the environment r, simply evaluate m in the environment r yielding the result a and then evaluate k a in the *same* environment r.

Lifting is easy, lift m is simply a constant function that ignores the environment.

As before, we define the class of environment monads. They can preform computations in environments using inEnv and read the current environment using rdEnv. We will see how is this used in the next example.

```
class Monad m => EnvMonad env m where
inEnv :: env -> m a -> m a
rdEnv :: m env
instance Monad m =>
EnvMonad (EnvT r m) where
inEnv r m = \_ -> m r
rdEnv = \r -> unit r
```

Now let's get back to our functional interpreter. We first define the Value type and then an instance of the InterpC class. We also have to make sure that our InterpM is an *environment error monad*. We can ensure that by using both ErrorT and EnvT monad transformers.

```
type Value = OR Int (OR Fun ())
type Fun = InterpM Value -> InterpM Value
```

```
instance InterpC TermF where
interp (Var v) = rdEnv 'bind' \env ->
case lookupEnv v env of
Just val -> val
Nothing ->
err("unbound variable: " ++ v)
```

```
interp (LambdaN s t) =
  rdEnv 'bind' \env ->
  unitInj(\arg->inEnv
                          (extendEnv(s,arg) env)
                          (interp t))
```

```
interp (LambdaV s t) =
  rdEnv 'bind' \env ->
  unitInj(\arg -> arg 'bind' \v ->
        inEnv (extendEnv(s,unit v) env)
               (interp t))
```

```
interp (App e1 e2) =
   interp e1 'bindPrj' \f ->
   rdEnv 'bind' \env ->
```

f(inEnv env (interp e2))

Interpreting a variable amounts to looking up its value in the environment. Both call-by-name and call-by-value abstractions return a function injected into the Value type and then embedded into the monad. The difference is that as soon as we apply the call-by-value version, it immediately reduces its argument, while the call-by-name version stores the unevaluated argument in the environment.

4.6 State monad transformer

State monad transformer adds state to a monad. Suppose that we want to introduce states of type s. Consider then the following type

type StateT s m a = s \rightarrow m (s,a).

We can view it as the type of stateful computations in m that result in a value of type a. A function of that type takes an old state as an argument and returns an m-computation of the new state and the result. We can use this as a base for our state transformer. As before, we first define the corresponding monadic transformation.

Unit computation of x simply returns x without changing the state. To evaluate m 'bind' k in the state s0, we first evaluate m in the state s0 yielding result a and the *new* state s1. Then we evaluate k a in the state s1.

Finally, we complete the definition of the state transformer by introducing the lift operator. Notice that lifted computations don't change the state, which is what we expect, since lifting shouldn't change the nature of "old" computations.

Naturally, state monads will be able to update the state. Our update member function will return the old state as a result and at the same time change the old state by applying an argument to it.

```
class Monad m => StateMonad s m where
 update :: (s -> s) -> m s
instance Monad m =>
        StateMonad s (StateT s m) where
 update f = \s -> unit (f s, s)
```

Notice that update id simply reads the current state. We can use the state monad transformer to implement the support for references. Locations will be represented as integers. Internal representation of the storage space is irrelevant. It is, however, important that we have the following functions available:

Our terms should be extended with reference constructs.

The semantics is obvious. To interpret Ref x, we first evaluate x, then allocate a new cell and store the result in it. The location of the new cell is returned as a result. Dereferencing simply does a lookup. Assignment updates the value stored at a given location and returns that new value as a result.

```
instance InterpC TermR where
interp (Ref x) =
    interp x 'bind' \val ->
    allocLoc 'bind' \loc ->
    updateLoc (loc, unit val) 'bind' \_->
```

```
unitInj loc
interp (Deref x) =
    interp x 'bindPrj' \loc ->
    lookupLoc loc
interp (Assign lhs rhs) =
    interp lhs 'bindPrj' \loc ->
    interp rhs 'bind' \val ->
    updateLoc (loc, unit val) 'bind' \_->
    unit val
```

Notice how the use of bindPrj checks whether a given computation computes a location and at the same time extracts the "raw" location number from it.

4.7 Lifting operations

In the preceding examples we saw that the nature of certain monads requires them to come equipped with some additional monadic operations besides unit and bind. For example, error monads have err :: String \rightarrow m a, environment monads have inEnv :: Env \rightarrow m a \rightarrow m a and state monads have update :: (s \rightarrow s) \rightarrow m s. What happens to those operations once we apply a monad transformer to our monad? They don't exist at the top level anymore. That seems to break the modularity of the original idea. What's the solution? Well, lift all the operations to the common top level! Unfortunately, the way to do that may sometimes be not so obvious and certainly not doable automatically. Let's state the problem precisely.

Suppose that for a given monad m, the set of types of (possibly monadic) operations is given by the following grammar:

$$\begin{aligned} \tau & ::= A & (type constants) \\ & | & a & (type variables) \\ & | & \tau \to \tau & (function types) \\ & | & (\tau, \tau) & (product types) \\ & | & m\tau & (monad types). \end{aligned}$$

For a given monad transformer t, we inductively define the mapping \prod_t that maps each type to its cor-

responding *lifted type* across the transformer t.

$$\begin{bmatrix} A \end{bmatrix}_t = A \\ \begin{bmatrix} a \end{bmatrix}_t = a \\ \begin{bmatrix} \tau_1 \to \tau_2 \end{bmatrix}_t = \begin{bmatrix} \tau_1 \end{bmatrix}_t \to \begin{bmatrix} \tau_2 \end{bmatrix}_t \\ \begin{bmatrix} (\tau_1, \tau_2) \end{bmatrix}_t = (\begin{bmatrix} \tau_1 \end{bmatrix}_t, \begin{bmatrix} \tau_2 \end{bmatrix}_t) \\ \begin{bmatrix} m & \tau \end{bmatrix}_t = t m \begin{bmatrix} \tau \end{bmatrix}_t$$

The problem of finding a *natural lifting* through the monad transformer t consists of finding for each type τ an operator

$$\mathcal{L}_{\tau} :: \tau \to [\tau]_t$$

such that the following conditions are met:

- 1. $\mathcal{L}_A = \mathrm{id}$
- 2. $\mathcal{L}_a = \mathrm{id}$
- 3. $\forall f. \ (\mathcal{L}_{\tau_1 \to \tau_2} f) \cdot \mathcal{L}_{\tau_1} = \mathcal{L}_{\tau_2} \cdot f$

This is shown on the following commuting diagram:

$$\begin{bmatrix} \tau_1 \end{bmatrix}_t \xrightarrow{\mathcal{L}_{\tau_1 \to \tau_2} I} \begin{bmatrix} \tau_2 \end{bmatrix}_t$$

$$\mathcal{L}_{\tau_1} \uparrow \qquad \uparrow \mathcal{L}_{\tau_2}$$

$$\tau_1 \xrightarrow{f} \qquad \tau_2$$

$$4. \ \mathcal{L}_{(\tau_1, \tau_2)} = \lambda(a, b). \ (\mathcal{L}_{\tau_1} a, \mathcal{L}_{\tau_2} b)$$

5. $\mathcal{L}_{m \tau} = \text{lift} \cdot (\text{map } \mathcal{L}_{\tau}).$

The above conditions ensure the naturality of lifting. Naturality is a-priori guaranteed just for lifting of pure computations, because of the requirements on the lift function. Here we see that the only problematic case is lifting in functional types. Liang, Hudak and Jones, besides giving a couple of examples, don't give any additional insight about how one might try to find such natural lifting. They deal with this problem in an ad-hoc manner. Here we propose a simple scheme that provides a way to find lifting operators systematically.

The idea is to try to find for each τ a pseudoinverse of \mathcal{L}_{τ} , i.e. an operator

$$\mathcal{U}_{\tau} :: [\tau]_t \to \tau$$

such that

$$\mathcal{U}_{\tau} \cdot \mathcal{L}_{\tau} = \mathrm{id.} \tag{1}$$

The operator \mathcal{U}_{τ} projects or unlifts lifted operations. Such \mathcal{U} can help us build \mathcal{L} and vice-versa. But first we have to realize such inversion on the basic levels. Formally, we look for a pseudoinverse

unlift ::
$$t \ m \ \tau \rightarrow m \ \tau$$

of the lifting operator

lift ::
$$m \ \tau \rightarrow t \ m \ \tau$$
.

Hence, we are looking for a function unlift such that

$$unlift \cdot lift = id. \tag{2}$$

Intuitively speaking, such operator will exist in all the cases where lift *injectively* embeds computations into the next layer.

Suppose we have such an operator. We can then construct \mathcal{L}_{τ} and \mathcal{U}_{τ} that satisfy (1) inductively for every type τ :

$$\begin{aligned} \mathcal{L}_A &= \mathrm{id} \\ \mathcal{U}_A &= \mathrm{id} \\ \mathcal{U}_A &= \mathrm{id} \\ \mathcal{L}_a &= \mathrm{id} \\ \mathcal{L}_{\tau_1 \to \tau_2} &= \lambda f \cdot \mathcal{L}_{\tau_2} \cdot f \cdot \mathcal{U}_{\tau_1} \\ \mathcal{U}_{\tau_1 \to \tau_2} &= \lambda f' \cdot \mathcal{U}_{\tau_2} \cdot f \cdot \mathcal{L}_{\tau_1} \\ \mathcal{L}_{(\tau_1, \tau_2)} &= \lambda(a, b) \cdot (\mathcal{L}_{\tau_1} a, \mathcal{L}_{\tau_2} b) \\ \mathcal{U}_{(\tau_1, \tau_2)} &= \lambda(a, b) \cdot (\mathcal{U}_{\tau_1} a, \mathcal{U}_{\tau_2} b) \\ \mathcal{L}_{m, \tau} &= \mathrm{lift} \cdot (\mathrm{map} \ \mathcal{L}_{\tau}) \end{aligned}$$

$$\mathcal{U}_m \ _{\tau} = (\mathrm{map} \ \mathcal{U}_{\tau}) \cdot \mathrm{unlift}$$

An easy inductive proof shows that (1) indeed holds. To show that the naturality conditions hold, we just need to verify it for the case of functional lifting, since other cases are trivial. Indeed, from the above definition we get

$$(\mathcal{L}_{\tau_1 \to \tau_2} f) \cdot \mathcal{L}_{\tau_1} = \mathcal{L}_{\tau_2} \cdot f \cdot \mathcal{U}_{\tau_1} \cdot \mathcal{L}_{\tau_1} = \mathcal{L}_{\tau_2} \cdot f,$$

since $\mathcal{U}_{\tau_1} \cdot \mathcal{L}_{\tau_1} = \text{id}.$

Here we give a couple examples. In the case of the environment transformer, we had

lift m =
$$r \rightarrow m$$
.

If r0 is any (fixed) environment, we can define a pseudoinverse by

unlift
$$l = 1 r0$$
.

Obviously, unlift (lift m) = m.

In the case of the state monad transformer, we had

lift $m = \langle s \rightarrow m$ 'bind' $\langle x \rightarrow unit(s,x) \rangle$.

Again, if s0 is an arbitrary state, we can define a pseudoinverse by

unlift
$$l = (l s0)$$
 'bind' $(s,x) \rightarrow unit(x)$.

Indeed, we get

(unlift.lift)(m) =
= unlift (\s -> m 'bind' \x -> unit(s,x)) =
= (m 'bind' \x -> unit(s0,x)) 'bind'
 \(s,x) -> unit(x) = /associativity/ =
= m 'bind'
 \x -> unit(s0,x) 'bind' \(s,x) -> unit(x) =
= m 'bind' \x -> unit(x) =
= m 'bind' unit = m.

4.8 Overview

Among the three approaches discussed in this paper, monad transformers seem to capture most of what was originally wished for in the most elegant way. Here are some of the main accomplishments:

- We can say that the Gofer type system with its constructor classes was definitely a hit. The key benefit comes from the usage of extensible union types that eliminates the need for Steele's towers of data types. Moreover, automatic inference of class instances and class methods (injections and projections, for instance) simplifies the the code and thus makes it less prone to errors.
- Monad transformers have the modularity of Steele's pseudomonads, but the code is smaller and more efficient. For example, in the case of monad transformers, we don't have to deal with

lists of routines (packages) at every level. Instead, we need to naturally lift the monadic operations across the monad transformers. This actually helps us develop a better understanding of the nature of operations we want to lift.

- What about generality? Monad transformers certainly generalize pseudomonads. Moreover, unlike in Steele's case, implementor is not limited with the fixed set of operations that a package can contain.
- Liang, Hudak and Jones spent less time on technical details (such as parsing). They concentrated on the essential ideas.

However, there are also some problems with this approach:

- Finding a natural lifting. The number of potential liftings grows quadratically with the number of monad transformers. No systematic procedure is presented, and even the simple schema described at the end of section 4.7 depends on finding a pseudoinverse unlift of the lift operator.
- When a sequence of monad transformers is applied to build an interpreter monad, the *order* in which we apply them matters a lot (rather than just the *set* of transformers). It is a non-trivial task to determine which order will yield the most efficient code or be the simplest one for implementing.
- Nonstandard environment. The Gofer, even though undoubtedly useful for this application, is not (yet) considered as a major Haskell standard. The author of this paper faced some serious noncompatibility problems while trying to run some examples.

5 Conclusions and related work

As shown in this paper, monads provide a very general framework for modelling programming language features. The core idea is to use a monad to define the notion of a *computation*. In the classical λ -calculus, we identify types of values and types of computations. By doing this, we fail to capture some interesting computational aspects, such as various kinds of *states* and control over the evaluator.

Monads can be used as an implementation tecnique, but also as a way to structure the language description. As Moggi pointed out in [6], traditional denotational semantics lacks modularity - it does not provide a general mechanism for looking at various language aspects separately. Monad transformers [5] can be used to achieve this in a monad setting. Moreover, they provide a good way of understanding the language features and their interaction and can even be used in the documentation of the language design. Philip Wadler [8] described how monads can be used to build interpreters. The idea is to construct one monad for each interpreter. This approach points to the right direction, but is not quite practical, since it does not scale - we have to rebuild our monad from start for each new feature added.

Guy Steele [3] tried to fix that by introducing a way to compose monads. For that purpose he used *pseu*domonads which are essentially "higher-order" monads (i.e. monads parametrized by monads). He developed a method of creating interpreter building blocks, each of which can implement a separate feature. Such blocks can then be glued together into a single interpreter. His approach failed in several aspects. First, he used a nonstandard notion of monads. Second, he struggled with some limitations of the Haskell type system which are circumvented through the use of his program simplifier. Actions of the program simplifier are not clearly defined which makes his system nonstandard, hard to use and possibly erroneous. Finally, he sacrificed generality in order to make the process of upgrading the interpreter more automatic. A further improvement came from S. Liang, P. Hudak and M. Jones [5] who also attacked the problem of monad compositionality, but with a better weapon - monad transformers. Monad transformers generalize pseudomonads and give a generally simple mechanism for building monads through multiple layers. Gofer's type system with constructor classes [1] proved to be very useful for this purpose. This approach, however, faced the problem of *lifting* of monadic operations to the top level. Only the correctness criteria for natural lifting is given, but the authors didn't provide any guideline about finding an actual lifting in a general case.

Different aspects of monads were studied by many other authors.

Jones and Wadler [2] showed how imperative features can be monadically imported to Haskell by designing a model which enables invocation of C-functions. This is a rather interesting combination, but its realization requires support for unboxed types which are somewhat unnatural in lazy programming languages. Launchbury and Jones [4] closely studied applications of *state transformers* for encapsulating different kinds of stateful computations in Haskell.

One may be wondering if there are any computational aspects at all that can not be modeled by monads?. Seems like the answer is (surprisingly?) yes! Philip Wadler [9] found that types of the certain kinds of continuations are too general to be interpreted in a monad. This looks like a good area for further research - perhaps we need some kind of generalized monads?

In general, we can conclude that monads are yet another witness to the close relationship between programming languages and category theory. It is important to investigate this connection throughly, since it may reveal some valuable discoveries in both areas. To what extent will this continue to happen in the future - remains to be seen.

References

- Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In In FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, 1993.
- [2] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Conference Record of POPL* '93, 1993.
- [3] Guy L Steele Jr. Building interpreters by composing monads. In Conference Record of POPL '94, 1994.

- [4] John Launchbury and Simon L Peyton Jones. State in haskell. J. of Lisp and Symbolic Computation, 1995.
- [5] Sheng Liang, Paul Hudak, and Mark Jones. Monad transforments and modular interpreters. In Conference Record of POPL '95, 1995.
- [6] Eugenio Moggi. An abstract view of programming languages. Technical report, University of Edinburgh, 1989.
- [7] Philip Wadler. Comprehending monads. Mathematical Structures in Computer Science, 2, 1992.
- [8] Philip Wadler. The essence of functional programming. In Conference Record of POPL '92, 1992.
- [9] Philip Wadler. Monads and composable continuations. J. of Lisp and Symbolic Computation, 1993.