Building Interpreters by Composing Monads

Guy L. Steele Jr.

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142
(617) 234-2860
`gls@think.com`

Abstract: We exhibit a set of functions coded in Haskell that can be used as building blocks to construct a variety of interpreters for Lisp-like languages. The building blocks are joined merely through functional composition. Each building block contributes code to support a specific feature, such as numbers, continuations, functions calls, or nondeterminism. The result of composing some number of building blocks is a parser, an interpreter, and a printer that support exactly the expression forms and data types needed for the combined set of features, and no more.

The data structures are organized as *pseudomonads*, a generalization of monads that allows composition. Functional composition of the building blocks implies type composition of the relevant pseudomonads.

Our intent was that the Haskell type resolution system ought to be able to deduce the approprate data types automatically. Unfortunately there is a deficiency in current Haskell implementations related to recursive data types: circularity must be reflected statically in the type definitions.

We circumvent this restriction by applying a purpose-built program simplifier that performs partial evaluation and a certain amount of program algebra. We construct a wide variety of interpreters in the style of Wadler by starting with the building blocks and a page of boilerplate code, writing three lines of code (one to specify the building blocks and two to (redundantly) specify type compositions), and then applying the simplifier. The resulting code is acceptable Haskell code.

We have tested a dozen different interpreters with various combinations of features. In this paper we discuss the overall code structuring strategy, exhibit several building blocks, briefly describe the partial evaluator, and present a number of automatically generated interpreters.

This is a preprint of a paper that is to appear in the Proceedings of the Twenty-first Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1994.

1 Introduction

I really liked Phil Wadler's work on monads [15, 16]. But I was not entirely satisfied with the methodology for constructing new interpreters by writing a single interpreter and then plugging in various monads. While it was not necessary to modify the code of the interpreter, it was necessary to alter "by hand" the data structure definitions for both the input expressions and the output values. It seemed to me that this could be automated.

Another problem was that, while there were many monads to choose from, they were one to a customer. You could have an interpreter with state, *or* an interpreter with continuations, but not an interpreter with continuations and state.

I had a vision of building blocks, like Legos or Tinkertoys, that could be used interchangeably and in combination to build interpreters. It seemed that if they could be combined by functional composition, then the Haskell [2] type system, or an extension of it, ought to be able to deduce the relevant data types automatically.

It can be done. With the function definitions presented in this paper, the value of the Haskell expression

```
complete (nondeterministic (cbv interpreter))
```

is in fact a complete nondeterministic call-by-value interpreter—indeed, not only an interpreter, but a parser and a printer as well. Similarly, the result of

```
complete (cbn (numbers interpreter))
```

is a complete call-by-name numbers interpreter; and the expression

```
complete (continuation (nondeterministic
  (cbv (numbers interpreter))))
```

produces a complete continuation nondeterministic call-by-value numbers interpreter.

2 Pseudomonads

We structure our interpreter building blocks by using *pseudomonads*, a generalization of monads that permits

```
infixl 4 &
infixl 1 <<, #, >>

type Unitfn p q = p -> q
type Bindfn p q = q -> (p -> q) -> q

type Pseudobindfn p q =     -- See text
  Monad q r -> q -> (p -> r) -> r

data Monad p q =
  Monad (Unitfn p q) (Bindfn 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

idmonad :: Monad p p
idmonad = Monad (\x -> x) (\z k -> k z)

data Pseudomonad p q =
  Pseudomonad (Unitfn p q) (Pseudobindfn p q)

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

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

data Bindtemp p q a b c = Bind2 (p, q)
                        | Bind3 (a, b, c)

x << y = Bind2 (x, y)

(Bind2 (x, y)) # z = Bind3 (x, y, z)

(Bind2 (x, y)) >> f = bind y x f
(Bind3 (x, y, z)) >> f = pseudobind z y x f

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

Figure 1: Haskell code for supporting pseudomonads

composition. The theory of pseudomonads is described in a companion paper [11]. The support code in Figure 1 is taken verbatim from that paper.

In order to convey necessary type information, it is convenient to *reify* monads and pseudomonads. We represent a monad in Haskell as an algebraic datatype `Monad` encapsulating the unit and bind operations for the monad. Note that we use the name `Monad` as both a type constructor and a data constructor, a convenient pun permitted by Haskell. We then define generic `unit` and `bind` operations.

In Wadler's style, one speaks of the type `a` and the type `M a` to which the type constructor `M` maps `a`; the monad in question is not itself an object of the language, but merely the conceptual triple (`M`, `unitM`, `bindM`). In our style, a monad is an object of type `Monad a b`, the type of monads that map type `a` to type `b`; thus `b` corresponds to Wadler's type `M a`.

As an example, the declaration of `idmonad` in Figure 1 defines the usual identity monad.

(A word about syntax: Wadler used Haskell infix notation for the various monad binding operations, writing `x 'bindM' f` instead of `bindM x f`. Our generic `bind` operation takes *three* arguments: a monad, an object, and a function. However, the infix notation is quite convenient, particularly for exhibiting the associative law for monads, so we have devised a kluge: we define two Haskell infix operators `<<` and `>>` so that we may write `bind m x f` as `x <<m>> f`. The code for this is also in Figure 1.)

A pseudomonad is only a slight generalization of a monad. A pseudomonad encapsulates two operations called `pseudounit` and `pseudobind`. The `pseudounit` operation is identical in its nature to `unit`, but the `pseudobind` operation takes an extra argument, before the object and the function to be applied, that is itself a monad. The idea is that `pseudobind` unwraps an object of type `q`, revealing an object of type `p`. This may then be fed to the function, but the function need not produce a result of type `q`; instead, it may produce a result of some other type `r`. The monad argument must be of type `Monad q r`, thus specifying a way to take values of type `q` to type `r`. The motivation is that the pseudobinding operation might not invoke the function after all, in which case it needs some other way to produce a result of type `r`.

(More about syntax: The generic `pseudobind` operation takes *four* arguments. We define `#` as yet another infix operator so that we may write `pseudobind p m x f` as `x <<m#p>> f`. This syntax happens to be used only once in the code figures in this paper, in the definition of `&` in Figure 1.)

In short, a pseudomonad is simply a monad that has been parameterized by another monad, in exactly the same manner that Wadler parameterized an interpreter by a monad.

Pseudomonads are assumed (and should be proved) to obey three laws analogous to those for monads. The monad laws, expressed in *unit*/*bind* form, are:

Left unit:

```
  unit m a <<m>> f  ≡  f a
```

Right unit:

```
  x <<m>> unit m  ≡  x
```

Associative:

```
      x <<m>> (\a -> f a  <<m>> g)
  ≡  (x <<m>>  \a -> f a) <<m>> g
```

(Remember that `<<...>>` has lower syntactic precedence than function application.)

The analogous laws for pseudomonads are:

Left unit:

```
unit p a <<m#p>> f  ≡  f a
```

Right unit:

```
x <<m#p>> (h . unit p)  ≡  h x
```

Associative:

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

The composition operator `m&p` composes a monad `m` and a pseudomonad `p` to produce what might be a new monad. If the monad obeys the three monad laws and the pseudomonad obeys the three pseudomonad laws, then the composition necessarily obeys the left unit and right unit monad laws. A separate proof must be supplied that the resulting monad obeys the monad associative law (and this is consistent with the experience of Moggi [6] and others that monads do not compose in general).

(We note in passing that there is in fact a more general theory of pseudmonads in which the first argument to a pseudobind operation is a pseudomonad rather than a monad. The composition operator `&` then combines two pseudomonads to produce a third pseudomonad:

```
pb&pa = Pseudomonad
          (pseudounit mb . pseudounit pa)
          (\pz x f -> x <<pz#pb>> (\w ->
                        w <<pz&pb#pa>> f))
```

This composition operator is associative (here we gloss over a subtle point about infinite regress in the proof) and its left and right identities are the identity pseudomonad. This more general theory is not required for the remainder of this paper but provided substantial theoretical motivation along the way.)

(But hold on, here! I have pulled a fast one! The type definition for `Pseudobindfn` in Figure 1 is not a legitimate Haskell type declaration; it has a free type variable. True enough, and that is entirely the point: that is the extra hole, the escape hatch that allows an arbitrary monad (or pseudomonad, in the more general theory) to be plugged in. One might try to fix the problem by introducing an extra type variable:

```
type Pseudobindfn p q r = ...
```

but this soon propagates throughout the code and becomes messy. Moreover, in the general theory it requires yet another type variable `s`:

```
type Pseudobindfn p q r =
   (Pseudomonad q r s) -> q -> (p -> r) -> r
```

and one gets caught in an infinite regress. This regress is part of the fundamental structure of the paradigm. Another idea is to wimp out and not say exactly what that first argument is:

```
type Pseudobindfn p q r x =
                        x -> q -> (p -> r) -> r
```

and hope that the Haskell type inference system will be happy with that; but it is not. The best solution is existential type variables; we would like to write the first definition shown and to have `r` treated existentially. I tried this out in the Chalmers Haskell implementation, which supports an experimental version of existential type variables. Unfortunately, that implementation imposes a restriction to the effect that existential type variables must not "escape"; but this application requires that they do escape. We recommend that Haskell implementors consider full support for unrestricted existential type variables. Our use of a program simplifier for the work reported here circumvents the restrictions of the Haskell type system.)

3 Towers of Data Types

An interpreter uses objects of two types: *terms* (input expressions) and *values* (results). When an interpreter is extended with a new capability, both of these types may require modification. For example, to add numeric processing to an interpreter, the value type must be extended to represent numbers and the term type must be extended to represent operations such as addition.

A building block takes an interpreter that maps 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 this through the use of two pseudomonads, one of type `Pseudomonad t t'` and one of type `Pseudomonad v v'`. The same pseudomonads are used to construct a new parser and printer. A parser maps strings into terms; a printer maps values into strings. The old parser is extended to produce terms of the new type `t'`. The old printer is extended to accept values of the new type `v'`.

Composing multiple building blocks results in composing multiple pseudomonads. The result is to construct two towers of data types. The final parser maps strings to the type at the top of the "term" tower; the final interpreter maps these terms to the type at the top of the "value" tower; and the final printer prints these values. See Figure 2.

So far, this is all fairly straightforward. The final fillip is that we wish both the final term type and the final value type to be recursive. For example, we want a term `(+` x y`)` to be able to contain *any* two terms x and y, and these terms must be of the topmost type in the tower. It is no problem to express this implicitly, but it causes a problem in the Haskell type checker. We deal with this problem later.

	term tower	value tower	
	T''	V''	
	$\vdots$	$\vdots$	
[*This figure is incomplete.*]	T_2	V_2	
[*See the proceedings.*]	T_1	V_1	
`String`	T_0	V_0	`String`

Figure 2: Towers of types

4 Packages

In effect, we need to compose not just functions but sets of functions. Haskell modules are not first-class in the language, so we had to invent some other representation for sets of functions. We use a simple association-list representation (a list of *routines*, where if there is more than one routine of the same type, the one earliest in the list shadows the others. We call this structure a *package*.

(We use an association list rather than, say, an 8-tuple for the sake of modularity; not all building blocks are concerned with all eight routine types. The association-list structure allows us to add new routine types without modifying all previously written building blocks.)

The necessary code for packages is shown in Figure 3. In this figure `t` and `v` stand for the "current" term and value types, that is, the types at the level of the type tower to which the package corresponds; `t''` and `v''` stand for the term and value types at the top of the tower. (The type `ve` stands for the type of values stored in the environment.) Thus the interpreter accepts a term at the current level of the type tower but produces a value at the top of the type tower. It does so because it produces a value at the current level and then, one way or another, applies the `unit` operation for a monad that will project the type to the top of the tower. (This monad is constructed as a composition of pseudomonads as the building blocks are composed.)

There are eight kinds of routines:

`ParseR`	the parser
`InterpR`	the interpreter
`ShowvalR`	the printer
`ComplainR`	signals an error
`MakenumR`	constructs a number value
`MakefunR`	constructs a function value
`ApplyR`	applies a function
`NameR`	a string that names the interpreter

Not every package will contain a routine of every kind. For example, there is no `MakenumR` routine unless the `numbers` building block has been included. These auxiliary routines provide a way for one building block to use facilities provided by another. For example, as we shall see, the `continuations` building block uses a `MakefunR` routine to construct a function representing a continuation; of course, it cannot do this unless the `cbv` or `cbn`

```
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'')
 | NameR String

data Package t v t'' v'' ve
 = Package [Routine t v t'' v'' ve]

update (Package pkg) new =
  Package (new ++ pkg)

parser (Package p) = parse' p where
  parse' (ParseR f:_) = f
  parse' (_:rest) = parse' rest

interpr (Package p) = interp' p where
  interp' (InterpR f:_) = f
  interp' (_:rest) = interp' rest

showvalr (Package p) = showval' p where
  showval' (ShowvalR f:_) = f
  showval' (_:rest) = showval' rest

complainr (Package p) = complain' p where
  complain' (ComplainR f:_) = f
  complain' (_:rest) = complain' rest

makenumr (Package p) = makenum' p where
  makenum' (MakenumR f:_) = f
  makenum' (_:rest) = makenum' rest

makefunr (Package p) = makefun' p where
  makefun' (MakefunR f:_) = f
  makefun' (_:rest) = makefun' rest

applyr (Package p) = apply' p where
  apply' (ApplyR f:_) = f
  apply' (_:rest) = apply' rest

namer (Package p) = name' p where
  name' (NameR x:_) = x
  name' (_:rest) = name' rest
```

Figure 3: Support code for packages

building block has been included to provide a definition of such a routine.

The `update` function adds a set of new routines to a package, possibly shadowing old ones of the same type. The various access functions `parser`, `interpr`, `showvalr`, etc., extract a routine of the appropriate type from a package; the result is undefined if the package contains no routine of the required type.

```
data TermZ = Bogon

data ValueZ = Wrong

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"
  complainZ s = unit tmv Wrong
  showvalZ Wrong = "<wrong>"
  nameZ = "interpreter"
```

Figure 4: The base interpreter prepackage

5 The Base Interpreter

The base interpreter, the one on which all variants are built and on which the towers of types are erected, is shown in Figure 4. It is a *null interpreter*. It interprets a completely boring language: the only term is `Bogon`. Every attempt to parse a string fails and produces a `Bogon`—actually not `Bogon`, but `Bogon` projected to the top of the term type tower. An attempt to interpret `Bogon` results in a complaint. Complaining produces the value `Wrong`—actually not `Wrong`, but `Wrong` projected to the top of the value type tower. The value `Wrong` prints as "`<wrong>`". The name of the base interpreter is "`interpreter`".

How are types projected to the top of the tower? The `interpreter` function takes two monads as arguments: `tmt` (the Top Monad for Terms) and `tmv` (the Top Monad for Values). These monads are constructed as building blocks are composed and then passed back down to all levels as arguments. The `unit` operations for thse monads are used by `parseZ` and `complainZ` to lift terms and values, respectively, to the top of the tower.

The `interpreter` function takes a third argument, which is also passed back down from the top of the tower. This is `top`, the completed top-level interpreter package. Note that `interpZ` does not call `complainZ` directly; it calls the complaint function of the top-level package. If any of the building blocks should shadow the complaint function with a new one, `interpZ` will use the new one, not `complainZ`.

6 Completing the Interpreter

The `interpreter` function shown in Figure 4 is not really a complete, working interpreter; it is merely a function that needs some arguments to produce a package of routines. We call such a function a *prepackage*; given

```
complete prepkg = top where
  top = prepkg idmonad idmonad top

type Term = TermZ                          --**
type Value = ValueZ                        --**
interp_pkg = complete interpreter          --**

parse = parser interp_pkg
interp = interpr interp_pkg
showval = showvalr interp_pkg
complain = complainr interp_pkg
makenum = makenumr interp_pkg
makefun = makefunr interp_pkg
apply = applyr interp_pkg
name = namer interp_pkg
```

Figure 5: Code to complete a constructed interpreter

two monads and a top-level package it will produce a new package (that may be at any point in the tower, possibly at the bottom or the top).

We need to feed three appropriate arguments to the prepackage. It is also handy to pull the constructed package apart into individual functions with the customary names. The necessary code is shown in Figure 5.

The function `complete` is deceptively simple. Given a prepackage, it simply caps the tower and returns the top-level package `top`. How is `top` computed? By feeding three arguments to the given prepackage. The first two arguments are simply the identity monad; whatever package is returned by the prepackage will already be at the top of the tower, so the identity monad suffices to lift terms and values of that package to the top. The third argument is `top`, the top-level package. It should now be apparent that we are depending critically on the fact that our implementation language, Haskell, is lazy. Computing the value of `top` requires that `top` be passed as an argument.

The result of the expression `complete interpreter` is in fact a complete (null) interpreter package. The remaining definitions simply name the individual routines in this package.

Finally, it is necessary to make a working Haskell main program. The driver code in Figure 6 implements a read-eval-print loop with an initial friendly greeting and a prompt before each interaction. Note that the `parse` function, according to the usual Haskell style, actually produces a list of possible parses; `read_eval_print` simply uses the first parse and discards any others.

When we apply our special-purpose program simplifier (more about this in Section 9) to all the code in Figures 1–5, the result is as shown in Figure 7.

Here is a sample interaction with this interpreter (all sample interactions in this paper are transcripts of actual console sessions with running code using the driver

```
data Term = Bogon

data Value = Wrong

parse s = [ (Bogon, s) ]

interp Bogon _ = complain "invalid expression"

complain s = Wrong

showval Wrong = "<wrong>"

name = "interpreter"
```

Figure 7: Simplified code for the completed base interpreter

of Figure 6):

```
Welcome to the interpreter!
> 3
<wrong>
> (2+3)
<wrong>
> \x.x
<wrong>
> Krazy Kat
<wrong>
```

Everything is `<wrong>`! How boring!

7 The Numbers Building Block

The `numbers` building block extends the term data type to include numeric constants `Con` n and addition operations `Add` x y, where n is a Haskell integer and x and y are top-level terms. It also extends the value data type to include numeric values `Num` n. See Figure 8.

In the definition of the type `TermN`, `t''` refers to the top-level term type and `t` refers to the term type at the next level down from the numbers package. Similarly for `v''` and `v` in the definition of `ValueN`.

The pseudomonad `mTN` is used to map from type `t` to the type `TermN t'' t`. It augments the type `t` with new possibilities `Con` n and `Add` x y for terms.

The pseudomonad `mVN` is used to map from type `v` to the type `ValueN v'' v`. It augments the type `v` with the new possibility `Num` n. (It may seem strange that `mVNbind` is not coded more simply as

```
mVNbind m (Num x) f = unit m (Num x)
mVNbind m (OtherVN x) f = f x
```

rather than using the intermediate name `qxfoo`. It is in fact strange and a kluge. The form in the figure tricks the program simplifier into applying a certain transformation at just the right time. It's a hack. I'm sorry.)

The function `numbers`, unlike `interpreter`, is *not* a prepackage; it is a building block. A building block

```
read_eval_print s = showval (interp exp []) where
  ((exp, _) : _) = parse s

main = appendChan stdout ("Welcome to the " ++ name ++ "!\n> ") abort
         (readChan stdin abort (\ contents -> process (lines contents))) where
           process [] = done
           process (x:xs) = appendChan stdout (read_eval_print x) abort
                              (appendChan stdout "\n> " abort
                                (process xs))
```

Figure 6: Code to drive a constructed interpreter

takes an old prepackage and produces a new prepackage. Thus `interpreter` is a suitable first argument for `numbers`; the result, a prepackage, is a suitable argument to `complete`. This prepackage accepts the usual three arguments `tmt`, `tmv`, and `top` and returns a new package. This package is produced by updating the old package with new routines—five of them, in this case.

And where did this "old package" come from? Ah, it must be constructed from the old prepackage originally given to the building block. The old prepackage will need three arguments. It needs a monad that will lift terms of the old package to the top. Well, `tmt` will lift terms from the *new* package to the top; and `mTN` will lift terms from the old package to the new package. All we need to do is compose `tmt` and `mTN` with the pseudomonad composition operator `&`. *Voilà!* Similarly for `tmv` and `mVN`. The top-level package `top` is passed down unchanged.

All this structure is quite stereotypical and appears more or less unchanged in every building block. Now let us examine the particulars of the `numbers` building block.

The parser is bulky (as parsers are wont to be, because they must distinguish bad inputs from good) but quite straightforward. There are three possibilities parse a constant *ddd* to produce the term `Con` n, parse an addition operation `(`x`+`y`)` to produce the term `Add` x y, or parse the way the old package parses. In the first two cases, the term is lifted to the top of the term type tower by applying `unit tmt`.

The interpreter `interpN` and printer `showvalN` are pretty much what you would expect after reading Wadler's paper [16]. Interpreting a constant `Con` n results in an equivalent value `Num` n, lifted to the top of the value type tower by applying `unit tmv`. The code for interpreting an `Add` operation is just like Wadler's, with `interpr top` in place of `interp`, `<<tmv>>` in place of `` `bindM` ``, and slightly more elaborate complaining when not both operands are numbers.

Finally, note that the name of the interpreter in the new package is constructed by prepending the word `numbers` to the name of the old package.

To incorporate the numbers facilities into an interpreter, all we need to do is replace the three lines marked "`**`" in Figure 5 with the following:

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

When we then apply our program simplifier to all the code in Figures 1–5 plus Figure 8, the result is as shown in Figure 9.

A sample interaction with this interpreter:

```
Welcome to the numbers interpreter!
> 3
3
> (2+3)
5
> \x.x
<wrong>
> Krazy Kat
<wrong>
> ((1+2)+(3+(4+5)))
15
```

It does indeed process numerical expressions and reject everything else.

8 Recursive Types in Haskell

The `complete` function is where the knot is tied, resulting in recursive term and value types; `top` is a fixpoint. Unfortunately, existing implementations of the Haskell type system choke on this circularity. The language specification is hazy, so it is hard to determine whether this is a language restriction or an implementation deficiency. The nub of the matter is that Haskell allows recursive and mutually recursive datatypes, provided that "an algebraic datatype intervenes" [2]. In current implementations this is apparently a static requirement; that is, any circularity of definition must be textually apparent rather than deduced by the type mechanism. If the program simplifier is not used, then the type checker discovers only dynamically for the type

```
data TermN t'' t = Con Int | Add t'' t'' | OtherTN t

data ValueN v'' v = Num Int | OtherVN v

mTN = Pseudomonad (\x -> 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 -> 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
    [ParseR parseN, InterpR interpN, ShowvalR showvalN, MakenumR makenumN, NameR nameN]
 where
  oldpkg = oldprepkg (tmt & mTN) (tmv & mVN) top
  parseN s = (pcon s ++ psum s ++ parser oldpkg s) where
    psum s = [(unit tmt (Add x y), s5) | ('(':s1) <- [dropWhile isSpace s],
                                          (x, s2) <- parser top s1,
                                          ('+':s3) <- [dropWhile isSpace s2],
                                          (y, s4) <- parser top s3,
                                          (')':s5) <- [dropWhile isSpace s4]]
    pcon (c:s) | (c >= '0' && c <= '9') = pcon' s (ord c - ord '0') where
      pcon' (c:s) n | (c >= '0' && c <= '9') = pcon' s (10*n + (ord c - ord '0'))
      pcon' s n = [(unit tmt (Con n), s)]
    pcon _ = []
  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))
                            (_, _) -> complainr top ("should be numbers: " ++
                                           showvalr top (unit tmv u) ++ ", " ++
                                           showvalr top (unit tmv v))
                          ))
  interpN (OtherTN x) env = interpr oldpkg x env
  showvalN (Num x) = show x
  showvalN (OtherVN x) = showvalr oldpkg x
  makenumN x = unit tmv (Num x)
  nameN = "numbers " ++ namer oldpkg
```

Figure 8: The numbers building block

`TermN t'' t` that in actual use `t` is `TermZ` and that `t''` is `TermN t'' t`. This last discovery produces an "occurs error" in both Chalmers Haskell and Glasgow Haskell, despite the fact that an algebraic datatype (dynamically) intervenes. The simplifier reduces the type declarations to a form in which the circularities are textually manifest (see Figure 9), thus rendering them palatable to these Haskell implementations.

9 The Program Simplifier

The principal activity of the program simplifier is judicious inlining of function definitions followed by β-reduction of both lambda-expressions and Haskell `case`-expressions. The simplifier also performs α-conversion where necessary and tries to do a smart job of it, renaming variables by adding primes; a post-pass heuristically tries to minimize the number of primes in the residual code while maintaining readability.

Ideally the necessary types could be deduced com-

```
data Term = Con Int | Add Term Term | Bogon

data Value = Num Int | Wrong

parse s = pcon s ++ psum s ++ [ (Bogon, s) ]
 where
  psum s' =
    [ (Add x y, s5)
    | ('(' : s1) <- [ dropWhile isSpace s' ],
      (x, s2) <- parse s1,
      ('+' : s3) <- [ dropWhile isSpace s2 ],
      (y, s4) <- parse s3,
      (')' : s5) <- [ dropWhile isSpace s4 ] ]
  pcon (c : s') | ((c >= '0') && (c <= '9')) =
    pcon' s' (ord c - ord '0') where
      pcon' (c' : s'') n
          | ((c' >= '0') && (c' <= '9')) =
        pcon' s''
              ((10 * n) + (ord c' - ord '0'))
      pcon' s'' n = [ (Con n, s'') ]
  pcon _ = []

interp (Con x) _ = Num x
interp (Add x y) env =
  case (interp x env, interp y env) of
    (Num j, Num k) -> Num (j + k)
    (_, _) ->
      complain ("should be numbers: " ++
                showval (interp x env) ++
                ", " ++
                showval (interp y env))
interp Bogon _ = complain "invalid expression"

makenum x = Num x

complain s = Wrong

showval (Num x) = show x
showval Wrong = "<wrong>"

name = "numbers interpreter"
```

Figure 9: Simplified code for the complete numbers interpreter

pletely automatically, but for this work we settled for performing substitution and simplification on explicitly provided type declarations for `Term` and `Value`. An interesting wrinkle is that when two algebraic datatypes are nested:

```
data Foo = Bar | Baz | OtherFoo Bletch
data Bletch = Quux | Ztesch
```

the type simplifier flattens them:

```
data Foo = Bar | Baz | Quux | Ztesch
```

and arranges for the code simplifier to eliminate applications of `OtherFoo` and to flatten nested `case` constructs as appropriate. (This transformation can be justified by an appeal to category theory; we omit the details here.)

A simple heuristic controls unrolling of recursive procedures: as the body of a procedure is inlined, any outermost `case` statement is tagged with the name of the procedure; and it is forbidden to inline within a `case` statement for a procedure name that matches the tag. Thus unrolling continues only if the gating `case` statement can first be eliminated by partial evaluation.

The simplifier uses a fast and fairly effective pretty-printer after the style of Waters [18, 12].

10 The Nondeterministic Building Block

The `nondeterministic` building block extends the term data type to include a choice construct `Amb` x y (surface syntax `(`$x|y$`)`), where x and y are top-level terms, and a failure operation `Fail` (surface syntax `fail`). It also alters the value data type to be a list of values. See Figure 10.

Once again the implementation of the interpreter closely follows the work of Wadler [16]. Failure results in a list of no value, but lifted to the top of value type tower. Choice involves interpreting each of the subexpressions and then appending the two lists of results. The difference here from Wadler's interpreter is that the values from the recursive calls to the interpreter are not necessarily lists; they are of the top value type and must be lowered back to the level of lists before concatenation. This is achieved by using the monad bind operation `<<tmv>>`. The bind operation requires that the result then be lifted back to the top value type (using `unit tmv`), which is exactly what we want anyway.

We choose to print the multiple values one per line, followed by the line "`That's all!`". The name of the interpreter is the name from the old package preceded by the word `nondeterministic`.

We can make a nondeterministic interpreter in exactly the same manner as we made a numbers interpreter; all we need do is to replace the three lines marked "`**`" in Figure 5 with the following:

```
type Term = TermL Term TermZ
type Value = ValueL Value ValueZ
interp_pkg = complete
  (nondeterministic interpreter)
```

However, it is even more interesting to make an interpreter that is nondeterministic *and* has numbers:

```
type Term = TermL Term (TermN Term TermZ)
type Value =
  ValueL Value (ValueN Value ValueZ)
interp_pkg = complete
  (nondeterministic (numbers interpreter))
```

```
data TermL t'' t = Fail | Amb t'' t'' | OtherTL t

type ValueL v'' v = [v]

mTL = Pseudomonad (\x -> OtherTL x) mTLbind where
  mTLbind m Fail f = unit m Fail
  mTLbind m (Amb x y) f = unit m (Amb x y)
  mTLbind m (OtherTL x) f = f x

mVL = Pseudomonad (\x -> [x]) mVLbind where
  mVLbind m x f = foldr c (unit m []) [ f a | a <- x ] where
    c x y = x <<m>> (\q -> y <<m>> (\r -> unit m (q ++ r)))

nondeterministic oldprepkg tmt tmv top = update oldpkg
    [ParseR parseL, InterpR interpL, ShowvalR showvalL, NameR nameL]
 where
  oldpkg = oldprepkg (tmt & mTL) (tmv & mVL) top
  parseL s = (pfail s ++ pchoice s ++ parser oldpkg s) where
    pfail s = [(unit tmt Fail, s1) | ('f':'a':'i':'l':s1) <- [dropWhile isSpace s]]
    pchoice s = [(unit tmt (Amb x y), s5) | ('(':s1) <- [dropWhile isSpace s],
                                            (x, s2) <- parser top s1,
                                            ('|':s3) <- [dropWhile isSpace s2],
                                            (y, s4) <- parser top s3,
                                            (')':s5) <- [dropWhile isSpace s4]]
  interpL Fail _ = unit tmv []
  interpL (Amb x y) env = interpr top x env <<tmv>> (\u ->
                          interpr top y env <<tmv>> (\v ->
                          unit tmv (u ++ v) ))
  interpL (OtherTL x) env = interpr oldpkg x env
  showvalL m = unlines [ showvalr oldpkg x | x <- m] ++ "That's all!"
  nameL = "nondeterministic " ++ namer oldpkg
```

Figure 10: The nondeterministic building block

When we then apply our program simplifier to all the code in Figures 1–5 plus Figures 8 and 10, the result is as shown in Figure 11. (To save space, we have chosen to elide the parser from this figure. The parsers follow a very predictable pattern anyway.)

A sample interaction with this interpreter:

```
Welcome to the nondeterministic numbers
interpreter!
> (2+3)
5
That's all!
> (2|3)
2
3
That's all!
> ((2+3)|(5+7))
5
12
That's all!
> ((2|3)|(5|7))
2
3
5
7
That's all!
> ((2|3)+(5|7))
7
9
8
10
That's all!
```

(Note that the two pairs of values interacted to produce four distinct sums.)

```
> (fail|5)
5
That's all!
> fail
That's all!
> ((2|3)+(fail|7))
9
10
That's all!
```

```
data Term = Fail | Amb Term Term
          | Con Int | Add Term Term | Bogon

type Value = [ Value' ]
data Value' = Num Int | Wrong

parse s = ...

interp Fail _ = []
interp (Amb x y) env =
  interp x env ++ interp y env
interp (Con x) _ = [ Num x ]
interp (Add x y) env =
  foldr (++)
        []
        [ foldr (++)
                []
                [ case (a, a') of
                    (Num j, Num k) ->
                      [ Num (j + k) ]
                    (_, _) ->
                      complain
                      ("should be numbers: "
                       ++
                       showval [ a ] ++
                       ", " ++
                       showval [ a' ])
                | a' <- interp y env ]
        | a <- interp x env ]
interp Bogon _ = complain "invalid expression"

makenum x = [ Num x ]

complain s = [ Wrong ]

showval m =
  unlines [ case x of
              Num x' -> show x'
              Wrong -> "<wrong>"
          | x <- m ] ++
  "That's all!"

name = "nondeterministic numbers interpreter"
```

Figure 11: Simplified code for the complete nondeterministic numbers interpreter (parser elided)

11 The Call-By-Value Building Block

It is of particular interest that what we usually think of as the very soul of an interpreter, the handling of variables and lambda-binding and function calls, is in fact just another set of features that can be separated out into a building block. The `cbv` building block extends the term data type to include variable references `Var` v, lambda expressions `Lam` v x, and function applications `App` x y, where v is a string and x and y are top-level terms. It also extends the value data type to include functional values `Fun` f, where f is a function; the function maps values at the current level of the type tower into values at the top of the type tower. See Figure 12.

Once again the implementation of the interpreter closely follows the work of Wadler [16]. However, we choose to nest the definition of `lookup` within the definition of one case of `interpCBV`.

Note that `makefunCBV` accepts a function `f` and composes it with `unit tmv`. This is because a call-by-value `Fun` value must be of type `v -> v''` but a `MakefunR` routine takes an argument of type `v'' -> v''`. There is a reason for this difference: as we will see, the call-by-name building block uses `Fun` values of a different type, but the `MakefunR` interface remains constants. This allows such clients of the `MakefunR` interface as the `continuation` building block to be combined with either the call-by-value or the call-by-name building block.

We parse the surface syntax "$\backslash v.x$" for lambda expressions and (f x) for application of function f to argument x. We print functional values as "`<function>`".

To add the call-by-value features into our nondeterministic numbers interpreter, we write this code:

```
type Term = TermCBV Term
  (TermL Term (TermN Term TermZ))
type Value = ValueCBV Value
  (ValueL Value (ValueN Value ValueZ))
interp_pkg = complete (cbv
  (nondeterministic (numbers interpreter)))
```

A sample interaction with this interpreter:

```
Welcome to the call-by-value nondeterministic
numbers interpreter!
> \x.x
<function>
> ((\x.\y.(x+y) 3) 4)
7
That's all!
> ((\f.\x.(f (f (f x))) \q.(q+q)) 3)
24
That's all!
```

Currying and functional arguments work fine.

```
> ((\x.\y.(x+y) (2|3)) (5|7))
7
9
8
10
That's all!
```

So far, so good ...

```
> ((\x.x|\y.(y+y)) 3)
3
That's all!
```

```
data TermCBV t'' t = Var String | Lambda String t'' | App t'' t'' | OtherTCBV t

data ValueCBV v'' v = Fun (ValueCBV v'' v -> v'') | OtherVCBV v

mTCBV = Pseudomonad (\x -> OtherTCBV x) mTCBVbind where
  mTCBVbind m (Var x) f = unit m (Var x)
  mTCBVbind m (Lambda v x) f = unit m (Lambda v x)
  mTCBVbind m (App x y) f = unit m (App x y)
  mTCBVbind m (OtherTCBV x) f = f x

mVCBV = Pseudomonad (\x -> OtherVCBV x) mVCBVbind where
  mVCBVbind m = qxfoo where
    qxfoo (Fun x) f = unit m (Fun x)
    qxfoo (OtherVCBV x) f = f x

cbv oldprepkg tmt tmv top = update oldpkg
    [ParseR parseCBV, InterpR interpCBV, ShowvalR showvalCBV,
     MakefunR makefunCBV, ApplyR applyCBV, NameR nameCBV] where
  oldpkg = oldprepkg (tmt & mTCBV) (tmv & mVCBV) top
  parseCBV s = (pvar s ++ plambda s ++ papp s ++ parser oldpkg s) where
    pvar (c:s) | c `elem` "abcdefghijklmnopqrstuvwxyz" = pvar' s [c] where
      pvar' (c:s) v | c `elem` "abcdefghijklmnopqrstuvwxyz" = pvar' s (v ++ [c])
      pvar' _ v = [(unit tmt (Var v), s)]
    pvar _ = []
    plambda s = [(unit tmt (Lambda v x), s4) | ('\\':s1) <- [dropWhile isSpace s],
                                               (Var v, s2) <- pvar s1,
                                               ('.':s3) <- [dropWhile isSpace s2],
                                               (x, s4) <- parser top s3]
    papp s = [(unit tmt (App x y), s5) | ('(':s1) <- [dropWhile isSpace s],
                                         (x, (c:s2)) <- parser top s1,
                                         isSpace c,
                                         s3 <- [dropWhile isSpace s2],
                                         (y, s4) <- parser top s3,
                                         (')':s5) <- [dropWhile isSpace s4]]
  interpCBV (Var v) env = lookup v env where
    lookup v ((w,z):_) | (v == w) = unit tmv z
    lookup v (_:e) = lookup v e
    lookup v [] = complainr top ("unbound variable: " ++ v)
  interpCBV (Lambda v x) env = unit tmv (Fun (\z -> interpr top x ((v,z):env)))
  interpCBV (App x y) env = interpr top x env <<tmv>> (\u ->
                            interpr top y env <<tmv>> (\v ->
                            applyr top u v ))
  interpCBV (OtherTCBV x) env = interpr oldpkg x env
  applyCBV (Fun f) x = f x
  applyCBV u _ = complainr top ("should be function: " ++
                                showvalr top (unit tmv u))
  showvalCBV (Fun x) = "<function>"
  showvalCBV (OtherVCBV x) = showvalr oldpkg x
  makefunCBV f = Fun (f . unit tmv)
  nameCBV = "call-by-value " ++ namer oldpkg
```

Figure 12: The call-by-value building block

That's strange ... why didn't it also print the value `6`?

```
> (\x.x|\y.(y+y))
<function>
```

Ooh, weird! Trying to choose a function produces only the first one! And it didn't even print "`That's all!`"! (It didn't print it when we tried "`\x.x`", either.)

But this is entirely correct. We chose to add in the call-by-value building block *after* adding in the nondeterministic building block. So the nondeterminism governs number values `Num` *n* but it doesn't govern functional values `Fun` *f*. Let's look at just the definition of the `Value` type produced by the program simplifier:

```
data Value = Fun (Value -> Value)
           | OtherVCBV [ Value' ]

data Value' = Num Int | Wrong
```

Sure enough, you can represent a list of numbers but not a list of functions.

The moral is that while you can compose any building blocks you like, it matters in what order you do it. This is not to say that the call-by-value nondeterministic numbers interpreter is "incorrect" in any sense; it is a perfectly legitimate, working combination of the specified building blocks into a working interpreter. It just may not be the interpreter we want for a given purpose. Perhaps we would prefer the nondeterministic call-by-value numbers interpreter:

```
type Term = TermL Term
  (TermCBV Term (TermN Term TermZ))
type Value = ValueL Value
  (ValueCBV Value (ValueN Value ValueZ))
interp_pkg = complete (nondeterministic
  (cbv (numbers interpreter)))
```

When we then apply our program simplifier to all the relevant code, the result is as shown in Figure 13. (Again we elide the parser.)

A sample interaction with this interpreter:

```
Welcome to the nondeterministic call-by-value
numbers interpreter!
> \x.x
<function>
That's all!
```

Ah, that's better!

```
> ((\x.\y.(x+y) 3) 4)
7
That's all!
> ((\f.\x.(f (f (f x))) \q.(q+q)) 3)
24
That's all!
> ((\x.\y.(x+y) (2|3)) (5|7))
7
9
8
10
That's all!
```

All this stuff works as before.

```
> ((\x.x|\y.(y+y)) 3)
3
6
That's all!
> (\x.x|\y.(y+y))
<function>
<function>
That's all!
```

And now we can also have choices of functions.

```
> (\x.(x+x) (2|3))
4
6
That's all!
```

This last exchange indicates that it is truly call-by-value: the two references to `x` in the body of `\x.(x+x)` both get the same value, either both `2` or both `3`.

12 The Call-By-Name Building Block

The call-by-name building block is quite similar to the call-by-value building block, with only some subtle differences in the use of the monads (the five lines differing other than in the names of variables from those in Figure 12 are marked `##`). But this makes all the difference in the world. See Figure 14.

We can specify the nondeterministic call-by-name numbers interpreter:

```
type Term = TermL Term
  (TermCBN Term (TermN Term TermZ))
type Value = ValueL Value
  (ValueCBN Value (ValueN Value ValueZ))
interp_pkg = complete (nondeterministic
  (cbn (numbers interpreter)))
```

(We forebear to show the simplified code; as you might expect, it differs in exactly five places from the simplified call-by-value code in Figure 13.)

A sample interaction with this interpreter:

```
Welcome to the nondeterministic call-by-name
numbers interpreter!
> \x.x
<function>
That's all!
> ((\x.\y.(x+y) 3) 4)
7
That's all!
```

```
data Term = Fail | Amb Term Term | Var String | Lambda String Term
          | App Term Term | Con Int | Add Term Term | Bogon

type Value = [ Value' ]
data Value' = Fun (Value' -> Value) | Num Int | Wrong

parse s = ...

interp Fail _ = []
interp (Amb x y) env = interp x env ++ interp y env
interp (Var v) env = lookup v env where
  lookup v' ((w, z) : _) | (v' == w) = [ z ]
  lookup v' (_ : e) = lookup v' e
  lookup v' [] = complain ("unbound variable: " ++ v')
interp (Lambda v x) env = [ Fun (\ z -> interp x ((v, z) : env)) ]
interp (App x y) env =
  foldr (++) [] [ foldr (++) [] [ apply a a' | a' <- interp y env ] | a <- interp x env ]
interp (Con x) _ = [ Num x ]
interp (Add x y) env =
  foldr (++) []
        [ case a of
            Fun x' -> [ Fun x' ]
            x' ->
              foldr (++) []
                    [ case a' of
                        Fun x'' -> [ Fun x'' ]
                        x'' ->
                          case (x', x'') of
                            (Num j, Num k) -> [ Num (j + k) ]
                            (_, _) ->
                              complain ("should be numbers: " ++
                                        showval [ x' ] ++
                                        ", " ++
                                        showval [ x'' ])
                    | a' <- interp y env ]
        | a <- interp x env ]
interp Bogon _ = complain "invalid expression"

apply (Fun f) x = f x
apply u _ = complain ("should be function: " ++ showval [ u ])

makefun f = Fun (f . (\ x -> [ x ]))

makenum x = [ Num x ]

complain s = [ Wrong ]

showval m =
  unlines [ case x of
              Fun x' -> "<function>"
              Num x' -> show x'
              Wrong -> "<wrong>"
          | x <- m ] ++
  "That's all!"

name = "nondeterministic call-by-value numbers interpreter"
```

Figure 13: Simplified code for the complete nondeterministic call-by-value numbers interpreter (parser elided)

```
data TermCBN t'' t = Var String | Lambda String t'' | App t'' t'' | OtherTCBN t

data ValueCBN v'' v = Fun (v'' -> v'') | OtherVCBN v                              --##

mTCBN = ...   -- identical to mTCBV except names have "N" instead of "V"

mVCBN = ...   -- identical to mVCBV except names have "N" instead of "V"

cbn oldprepkg tmt tmv top = update oldpkg
    [ParseR parseCBN, InterpR interpCBN, ShowvalR showvalCBN,
     MakefunR makefunCBN, ApplyR applyCBN, NameR nameCBN] where
  oldpkg = oldprepkg (tmt & mTCBN) (tmv & mVCBN) top
  parseCBN s = ...  -- identical to parseCBV
  interpCBN (Var v) env = lookup v env where
    lookup v ((w,z):_) | (v == w) = z                                             --##
    lookup v (_:e) = lookup v e
    lookup v [] = complainr top ("unbound variable: " ++ v)
  interpCBN (Lambda v x) env = unit tmv (Fun (\z -> interpr top x ((v,z):env)))
  interpCBN (App x y) env = interpr top x env <<tmv>> (\u ->
                            applyr top u (interpr top y env))                     --##
  interpCBN (OtherTCBN x) env = interpr oldpkg x env
  applyCBN (Fun f) x = f x
  applyCBN u _ = complainr top ("should be function: " ++
                                showvalr top (unit tmv u))
  showvalCBN (Fun x) = "<function>"
  showvalCBN (OtherVCBN x) = showvalr oldpkg x
  makefunCBN f = unit tmv (Fun f)                                                 --##
  nameCBN = "call-by-name " ++ namer oldpkg                                        --##
```

Figure 14: The call-by-name building block (some parts identical to those in call-by-value building block elided)

```
> ((\f.\x.(f (f (f x))) \q.(q+q)) 3)
24
That's all!
> ((\x.\y.(x+y) (2|3)) (5|7))
7
9
8
10
That's all!
> ((\x.x|\y.(y+y)) 3)
3
6
That's all!
> (\x.x|\y.(y+y))
<function>
<function>
That's all!
```

This is all exactly as before. How do we know it is call-by-name?

```
> (\x.(x+x) (2|3))
4
5
5
6
That's all!
```

This last exchange indicates that it is truly call-by-name: the two references to `x` in the body of `\x.(x+x)` each perform the calculation `(2|3)` anew, resulting in a total of four possibilities.

13 The Continuation Building Block

The `continuation` building block extends the term data type to include a construct `Catch` *v y*. (Wadler [16] called this construct `Callcc`, but this was an error; it has the form of the original Scheme `CATCH` construct [14]. However, his function `callccK` was indeed a "callcc"-type operation.) The `continuation` building block also alters the value data type to be a Haskell function that, when given a *continuation* (a Haskell function that maps values to values) produces a value. See Figure 15.

The `continuation` building block was the most difficult to construct—it took a *long* time to get the types right! Even so, problems lurk. We would much prefer that the value type be

```
type ValueC v'' v = ((v -> r) -> r)
```

where `r` is a free type variable. In other words, we want the lifted value type to be polymorphic over various types of continuation that produce various types of result, for there are conflicting requirements. In `showvalC`, we need to be able to extract an underlying value by feeding the identity continuation to a `ValueC` value; here we need `r` = `v`. But the implementation of `Catch` shown in Figure 15 requires that a "captured continuation" return a value at the top of the type tower; here we need `r` = `v''`. The net effect is that we must have `v` = `v''`. The practical effect is to limit the `continuation` building block to be the *last* (that is, *leftmost*) one applied. (If you try to build a nondeterministic continuation interpreter, for example, Haskell will report a type error on the simplified code; it simply doesn't work out. A continuation nondeterministic interpreter works fine.)

We can specify the continuation nondeterministic call-by-value numbers interpreter:

```
type Term = TermC Term (TermL Term
  (TermCBV Term (TermN Term TermZ))
type Value = ValueC Value (ValueL Value
  (ValueCBV Value (ValueN Value ValueZ))
interp_pkg = complete (continuation
  (nondeterministic (cbv
    (numbers interpreter))))
```

When we apply our program simplifier to all the relevant code, the result is as shown in Figure 16. (This time we elide the parser *and* the interpreter case for `Add`.)

Inspection of the interpreter case for `Catch` reveals the expected code: `x` is interpreted in an environment in which variable `v` is bound to a function that, when given a value `z` and a continuation `cc`, proceeds to ignore `cc` and instead feeds to `z` the continuation `q` of the `Catch` expression. (Note that the simplifier wasn't smart enough to do an η-reduction on `\a -> q a`.)

It is also of interest to examine the first argument to each occurrence of `foldr` in the interpreter case for `App`. Recall that the call to `foldr` comes from the `nondeterministic` building block; its purpose is to append-reduce (in monad terminology, *join*) a list of lists. Now that we have added in the `continuation` building block, we see that the first argument to `foldr` is none other than the continuation-passing version of `++`! And its second argument is the continuation-passing version `\c -> c []` of the empty list `[]`

A sample interaction with this interpreter:

```
Welcome to the continuation nondeterministic
call-by-value numbers interpreter!
> Catch v (2+3)
5
That's all!
> Catch v (2+(v 3))
3
That's all!
> (4+Catch v (2+(v 3)))
7
That's all!
> (4+Catch v (2|(v 3)))
7
That's all!
> (4|Catch v (2|(v 3)))
4
3
That's all!
```

14 The Errors Building Block

The `errors` building block introduces no new terms; its only purpose is to preserve the string argument given to the `complain` function rather than discarding it, thereby producing an error message perhapos more useful than "`<wrong>`"!

```
Welcome to the nondeterministic errors
call-by-value numbers interpreter!
> (3+x)
Error: unbound variable: x
That's all!
> (3+(x|(5|y)))
Error: unbound variable: x
8
Error: unbound variable: y
That's all!
> (3+(x|(|y)))
Error: unbound variable: x
Error: invalid expression
Error: unbound variable: y
That's all!
> (3+(x|\y.y))
Error: unbound variable: x
<function>
That's all!
```

The last interaction reminds us of an unsatisfactory property of the interpreter framework we have used. Because functions are higher in the value tower than numbers, attempting to add a function to a number does not produce the error message "`should be numbers` ..." but merely returns the function (as dictated by the pseudobind operation `mVCBVbind` in Figure 12). (A possible solution: the code in `interpN` for `Add` should not use pseudobinding to extract a numeric value, but a related operation that calls an "error thunk" if it is unable to descend to the `Num` representation. The problem is that this operation must be provided by all pseudomonads; but it may be worth it.)

```
data TermC t'' t = Catch String t'' | OtherTC t

type ValueC v'' v = ((v -> v) -> v)

mTC = Pseudomonad (\x -> OtherTC x) mTCbind where
  mTCbind m (Catch v x) f = unit m (Catch v x)
  mTCbind m (OtherTC x) f = f x

mVC = Pseudomonad (\x c -> c x) mVCbind where
  mVCbind m x f cc = x (\a -> f a cc)

continuation oldprepkg tmt tmv top = update oldpkg
      [ParseR parseC, InterpR interpC, ShowvalR showvalC, NameR nameC] where
  oldpkg = oldprepkg (tmt & mTC) qxtmv top
  qxtmv = tmv & mVC
  parseC s = (pcatch s ++ parser oldpkg s) where
    pcatch s = [(unit tmt (Catch v x), s3)
               | ('C':'a':'t':'c':'h':c1:s1) <- [dropWhile isSpace s],
                 isSpace c1,
                 (Var v, c2:s2) <- parser top s1,
                 isSpace c2,
                 (x, s3) <- parser top s2]
  interpC (Catch v x) env =
    callcc (\k -> interp x ((v, makefunr top (\z -> z <<qxtmv>> k)):env)) where
      callcc h q = h (\b d -> q b) q
  interpC (OtherTC x) env = interpr oldpkg x env
  showvalC x = showvalr oldpkg (x (\q -> q))
  nameC = "continuation " ++ namer oldpkg
```

Figure 15: The continuation building block

15 Why Haskell?

The restrictions of the Haskell type system certainly caused some problems along the way. Several times I was sorely tempted to switch to Scheme or Common Lisp. Had I used a dialect of Lisp, it would have been much easier to code the program simplifier, because Lisp already provides a parser and pretty-printer. Then again, working in Lisp might have avoided the need for a program simplifier altogether, since the principal practical motivation for it was to transform the code into a form acceptable to the Haskell type-checker.

The truth is that (a) Wadler had conducted his work in Haskell and it seemed appropriate to preserve notational continuity, and (b) I was looking for an excuse to gain experience with using Haskell anyway. In the end, I have to say that the type checking was more help than hindrance, especially in the construction of the `continuations` building block. I had the same experience with Haskell that I had twenty years ago with ECL [19] (which was, in effect, also a strongly-typed dialect of Lisp): almost always, once I made the type checker happy, the program was correct. Moreover, the challenge of trying force pseudomonads into the Haskell type framework taught me a lot about the strengths and limitations of both, in much the same way that trying to hammer a set of lovely but vague thoughts into the form of a sonnet yields great insight into the strengths and limitations of the English language (for example, the wealth of synonyms but the paucity of rhymes for "love") while clarifying, not always comfortably, one's romantic impulses.

16 Why Monads?

More than one reviwer of this work has asked me, "So what's the big deal about monads? Isn't what you have done simply 'good functional programming style'?" I feel compelled to explain myself on this point.

Well, maybe that's all it is, "merely" good style (which perhaps I should simply accept as a compliment, never mind that it may be back-handed!). But I feel that I never would have thought of this style of decomposing interpreters without exposure to the structuring ideas suggested by monads and the use to which Philip Wadler, Mark Jones, and others have put them. Theory is not automatically wonderful by virtue of being theory, but it justifies itself by leading one to pragmatic solutions that would otherwise have seemed implausible. That is exactly what happpened in this instance.

```
data Term = Catch String Term | Fail | Amb Term Term | Var String
          | Lambda String Term | App Term Term | Con Int | Add Term Term | Bogon

type Value = ([ Value' ] -> [ Value' ]) -> [ Value' ]
data Value' = Fun (Value' -> Value) | Num Int | Wrong

parse s = ...

interp (Catch v x) env = \ q ->
  interp x ((v, makefun (\ z -> \ cc -> z (\ a -> q a))) : env) q
interp Fail _ = \ c -> c []
interp (Amb x y) env = \ cc -> interp x env (\ a -> interp y env (\ a' -> cc (a ++ a')))
interp (Var v) env = lookup v env where
  lookup v' ((w, z) : _) | (v' == w) = \ c -> c [ z ]
  lookup v' (_ : e) = lookup v' e
  lookup v' [] = complain ("unbound variable: " ++ v')
interp (Lambda v x) env = \ c -> c [ Fun (\ z -> interp x ((v, z) : env)) ]
interp (App x y) env = \ cc ->
  interp x env
         (\ a ->
            foldr (\ x' y' -> \ cc' -> x' (\ a' -> y' (\ a'' -> cc' (a' ++ a''))))
                  (\ c -> c [])
                  [ \ cc' ->
                      interp y env
                             (\ a'' ->
                                foldr (\ x' y' ->
                                         \ cc'' ->
                                           x' (\ a''' -> y' (\ a'''' -> cc'' (a''' ++ a''''))))
                                      (\ c -> c [])
                                      [ apply a' a''' | a''' <- a'' ]
                                      cc')
                  | a' <- a ]
                  cc)
interp (Con x) _ = \ c -> c [ Num x ]
interp (Add x y) env = \ cc -> ...
interp Bogon _ = complain "invalid expression"

apply (Fun f) x = f x
apply u _ = complain ("should be function: " ++ showval (\ c -> c [ u ]))

makefun f = Fun (f . (\ x c -> c x) . (\ x -> [ x ]))

makenum x = \ c -> c [ Num x ]

complain s = \ c -> c [ Wrong ]

showval x =
  unlines [ case x' of
              Fun x'' -> "<function>"
              Num x'' -> show x''
              Wrong -> "<wrong>"
          | x' <- x (\ q -> q) ] ++
  "That's all!"

name = "continuation nondeterministic call-by-value numbers interpreter"
```

Figure 16: Simplified code for the complete continuation nondeterministic call-by-value numbers interpreter (parser and interpreter case for `Add` elided)

```
type TermE t'' t = t

data ValueE v'' v = Err String | OtherVE v

mTE = Pseudomonad (\x -> x) (\x f -> f x)

mVE = Pseudomonad (\x -> OtherVE x) mVEbind where

  mVEbind m = parsequux where
    parsequux (Err x) f = unit m (Err x)
    parsequux (OtherVE x) f = f x

errors oldprepkg tmt tmv top = update oldpkg
  [ShowvalR showvalE, ComplainR complainE,
   NameR nameE] where
  oldpkg = oldprepkg (tmt & mTE) (tmv & mVE) top
  complainE s = unit tmv (Err s)
  showvalE (Err x) = "Error: " ++ x
  showvalE (OtherVE x) = showvalr oldpkg x
  nameE = "errors " ++ namer oldpkg
```

Figure 17: The errors building block

(I draw a parallel to the changes in style of an assembly language programmer once exposed to the principles of structured programming.) Moreover, I hold out some hope that, once the slightly ad hoc techniques I have stumbled across here have been back-translated into appropriately mathematical language, some suitably qualified category theorist will remark, “Oh, yes, that's similar to this notion we've known about for years; it has the following marvelous properties, and if you will only do things *this* way instead you'll have a much easier time of it.” Which is an excellent relation for theory to hold to practice.

17 Comparison with Other Work

This paper is essentially one long reformulation of Wadler's work [16] to add the crucial capability of composition. It owes its roots to all the work that Wadler cites, of course, notably that of Reynolds [10]. It is also in the spirit of my own earlier work with Sussman [13].

In this presentation I have omitted all the proofs, particularly proofs of monad associativity. In some cases they do not exist; that is, some of the putative monads that arise in the interpreters presented here are not really associative. This obstructs proofs of program equivalence of the kind discussed by Wadler [16] but otherwise seems to have little practical effect. Nevertheless, further investigation is warranted. Certain simple but important cases, such as lists and simple coproducts, are well-behaved; when provided as pseudomonads to the composition operator `&` the result obeys the monad associative law and therefore really is always a monad. King and Wadler [5], for example, consider not only lists but also trees, bags, and sets.

Simon Peyton Jones and Wadler have investigated the use of monads to perform I/O and other “imperative” tasks within a working compiler coded in Haskell [7].

Jonathan Rees has coded monadic interpreters in Scheme but this work apparently has not been published [9].

Mark Jones has developed Gofer, a variant of Haskell that allows additional polymorphism through the use of type classes [4]. Gofer supports programming with multiple monads, including a cleaner version of the monad comprehension syntax proposed by Wadler [15]. Representing monads as a Gofer type class allows the Gofer type system to deduce which monad is intended in a unit or bind operation, avoiding the need to reify monads in the manner shown in this paper. Jones has begun to explore the possibility of recoding the interpreters presented here so as to use Gofer's simpler monadic programming style [3].

Many of the ideas presented here were anticipated by Moggi [6]:

> To give semantics to a complex language L we propose a stepwise approach, which starts from a monad (notion of computation) corresponding to a toy sublanguage of L and then at each step applies a *monad constructor* which adds one feature to the language.

Moggi then goes on to present theoretical descriptions of several useful monad constructors, including exceptions, side effects, and continuations. To the best of our knowledge, however, the work presented here is the first to put this idea into practice to produce actual working interpreters from monad-like building blocks.

Our work also factors Moggi's monad constructors in an interesting way into two parts: a pseudomonad and the composition operator `&`. The composition operator is a nonvarying associative operator with left and right identity; a pseudomonad is almost identical in structure to a monad. If `p` is a pseudomonad, then `(&p)` (which in Haskell means the same as `\x -> (x&p)`) is a monad constructor, mapping a monad into a new monad. I think that this factoring usefully structures and simplifies the necessary proofs that a composed monad will satisfy the monad laws.

18 Conclusions and Future Work

The title of this paper slyly plays on an ambiguity in the use of the English verb "to compose": the object of the verb may indicate either the *inputs* or the *outputs* of the composition operation. Indeed, you can't really compose (i.e., *combine*) two monads and get a monad as the result. But you can compose (i.e., *produce*) a monad from two other things (not both of which are monads). Our technique does compose monads in this sense by applying a composition operator to pseudomonads.

An important lesson from this work is that the composition operation is associative but in general not commutative. The practical result is that it really matters in what order you combine the building blocks. Two interpreters built from the same building blocks combined in different orders may have different (perhaps mysteriously different) behaviors.

It should be clear that, even using only the small set of building blocks presented here, we can automatically construct more interpreters than you can shake a stick at. (I was sorely tempted to title this paper "Interpreters for Free!" [17] but thought better of it.) I have constructed many more interpreters than I have shown here and am working on other building blocks, for state, output, and string data, for example.

The building blocks shown here transform the value type in a variety of ways but all do the same thing to the term type, adding a few alternatives to one big disjunction. What if some pseudomonad did some nontrivial computation on terms? Presto! We have macros. There should be a `macros` building block. It would leave the value type alone but transform the term data type, perhaps according to some macro environment.

How about error messages with positions? Wadler [16] remarks, "The parser will produce `[At]` terms as suitable." So the term type must be extended, but it turns out that it is easiest to modify the parser so that its input data type is not merely a string but a pair of a string (remaining to be parsed) and a position (of the start of that remaining string). So it is useful to have more than two type towers. We should have left a hook [8] so that the input type to the parser could be changed.

Indeed, it is unsatisfactory that the `interp` functions shown here must always accept an extra environment argument `env`. Environments are irrelevant in the `numbers` building block, for example, or in the `nondeterministic` building block; they ought to be introduced explicitly by the `cbv` and `cbn` building blocks, for only they are explicitly concerned with environments. An interpreter ought to be purely a mapping from terms to values, with no extraneous arguments. However, it does not suffice simply to work within the value tower, extending it so that the call-by-value building block lifts the value type `v` to `env -> v`. The problem is that if further building blocks are composed, such as `nondeterministic`, the environments will not be "inherited" properly. One solution seems to be to introduce an additional hook in the form of a second term type tower, so that one term type represents the parser output, the second represents the interpreter input, and the main driver calls a small function (initially the identity function) to map from one to the other. We suspect that a full-blown, truly modular Lisp interpreter coded in this style would need a fairly large number of types with corresponding towers.

The techniques presented here are quite general and need not be limited to the construction of interpreters; interpreters are merely a good subject for exposition in academic papers because their behavior is relatviely complex relative to the size of code needed to express it. Monads (and pseudomonads) are likely to be useful in any software system that requires easy and flexible extensibility.

The use of a monad is a hook, a powerful hook, a hook with *type discipline*. A monad has an eye; when you hang a monad on the hook, it uses up the hook. A pseudomonad has an eye on one end and a hook on the other. You can chain them. (Hm. I'm mixing my metaphors. First towers, now chains.) When you hang a pseudomonad on a hook, there's still a hook left to use. I wonder whether the Macintosh toolbox [1] could be defined in terms of a large set of pseudomonads, lending some structure to the games played by various add-on software packages to intercept system calls?

19 Acknowledgments

I wish to thank Philip Wadler, Mark Jones, Olivier Danvy, Paul Hudak, and the conference referees for their helpful remarks. Paul Hudak, Phil Wadler, Simon Peyton Jones, and Lennart Augustsson provided helpful advice on the use of Haskell. I am grateful for access to four different implementations of Haskell: Yale Haskell, Glasgow Haskell, Chalmers Haskell, and Jones' Gofer system. Jonathan Rees, Dan Friedman, and Mitch Wand also provided useful information and encouragement.

References

[1] Apple Computer, Inc. *Inside Macintosh* (five volumes). Addison-Wesley (Reading, Massachusetts, 1985–86).

[2] Hudak, Paul, Peyton Jones, Simon, and Wadler, Philip, editors. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language (Version 1.1).* Technical Report. Yale University and Glasgow University (New Haven and Glasgow (respectively), August 1991).

[3] Jones, Mark P. Personal communication to Guy Steele, September 1993.

[4] Jones, Mark P. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proc. FPCA '93: The Sixth International Conference on Functional Programming Languages and Computer Architecture.* ACM SIGPLAN/SIGARCH and IFIP (London, September 1993).

[5] King, David J., and Wadler, Philip. Combining monads. In *Functional Programming, Glasgow '92.* Springer Verlag (Berlin, 1992).

[6] Moggi, Eugenio. *An Abstract View of Programming Languages.* Technical Report ECS-LFCS-90-113. Laboratory for Foundations of Computer Science, University of Edinburgh (Edinburgh, Scotland, April 1990). Lecture notes for a course taught at Stanford University, Spring 1989.

[7] Peyton Jones, Simon L., and Wadler, Philip. Imperative functional programming. In *Proc. Twentieth Annual ACM Symposium on Principles of Programming Languages.* Association for Computing Machinery (Charleston, South Carolina, January 1993), 1–14.

[8] Raymond, Eric, editor. *The New Hacker's Dictionary.* MIT Press (Cambridge, Massachusetts, 1991).

[9] Rees, Jonathan. Personal communication to Guy Steele, October 1993.

[10] Reynolds, John C. Definitional interpreters for higher order programming languages. In *Proc. ACM National Conference.* Association for Computing Machinery (Boston, August 1972), 717–740.

[11] Steele, Guy L., Jr. *How to Compose Monads.* Technical Report. Thinking Machines Corporation (Cambridge, Massachusetts, July 1993). Unpublished.

[12] Steele, Guy L., Jr., Fahlman, Scott E., Gabriel, Richard P., Moon, David A., Weinreb, Daniel L., Bobrow, Daniel G., DeMichiel, Linda G., Keene, Sonya E., Kiczales, Gregor, Perdue, Crispin, Pitman, Kent M., Waters, Richard C., and White, Jon L. *Common Lisp: The Language (Second Edition).* Digital Press (Bedford, Massachusetts, 1990).

[13] Steele, Guy Lewis, Jr., and Sussman, Gerald Jay. *The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two).* AI Memo 453. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, May 1978).

[14] Sussman, Gerald Jay, and Steele, Guy Lewis, Jr. *SCHEME: An Interpreter for Extended Lambda Calculus.* AI Memo 349. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, December 1975).

[15] Wadler, Philip. Comprehending monads. In *Proc. 1990 ACM Symposium on Lisp and Functional Programming.* ACM SIGPLAN/SIGACT/SIGART (Nice, France, June 1990), 61–77. To appear in the journal *Mathematical Structures in Computer Science.*

[16] Wadler, Philip. The essence of functional programming. In *Proc. Nineteenth Annual ACM Symposium on Principles of Programming Languages.* Association for Computing Machinery (Albuquerque, New Mexico, January 1992), 1–14.

[17] Wadler, Philip. Theorems for free! In *Proc. FPCA '89: The Fourth International Conference on Functional Programming Languages and Computer Architecture.* ACM SIGPLAN/SIGARCH and IFIP (London, September 1989), 347–359.

[18] Waters, Richard C. *XP: A Common Lisp Pretty Printing System.* AI Memo 1102. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, March 1989).

[19] Wegbreit, Ben, Holloway, Glenn, Spitzen, Jay, and Townley, Judy. *ECL Programmer's Manual.* Technical Report 23-74. Harvard University Center for Research in Computing Technology (Cambridge, Massachusetts, December 1974).