Programmng and Reasoning with Algebraic Effects and Dependent Types

Edwin Brady
School of Computer Science, University of St Andrews, St Andrews, Scotland.
Email: ecb10@st-andrews.ac.uk

Abstract
One often cited benefit of pure functional programming is that pure code is easier to test and reason about, both formally and informally. However, real programs have side-effects including state management, exceptions and interactions with the outside world. Haskell solves this problem using monads to capture details of possibly side-effecting computations — it provides monads for capturing state, I/O, exceptions, non-determinism, libraries for practical purposes such as CGI and parsing, and many others, as well as monad transformers for combining multiple effects.

Unfortunately, useful as monads are, they do not compose very well. Monad transformers can quickly become unwieldy when there are lots of effects to manage, leading to a temptation in larger programs to combine everything into one coarse-grained state and exception monad. In this paper I describe an alternative approach based on handling algebraic effects, implemented in the Idris programming language. I show how to describe side-effecting computations, how to write programs which compose multiple fine-grained effects, and how, using dependent types, we can use this approach to reason about states in effectful programs.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) Languages

General Terms Languages, Verification

Keywords Dependent Types, Algebraic Effects

1. Introduction
Pure functions do not have side effects, but real applications do: they may have state, communicate across a network, read and write files, or interact with users, among many other things. Furthermore, real systems may fail due to corrupted or untrusted data. Pure functional programming languages such as Haskell [26] manage such problems using monads [31], and allow multiple effects to be combined using a stack of monad transformers [17].

Monad transformers are an effective tool for structuring larger Haskell applications. An application using a Read-Execute-Print loop, for example, may have some global state and perform console I/O, and hence be built with an IO monad transformed into a state monad using the StateT transformer. However, this can create difficulties: the order in which transformers are applied matters (that is, transformers do not commute easily); and it is difficult to invoke a function which uses a subset of the transformers in the stack. To illustrate these problems, consider an evaluator, in Haskell, for a simple expression language:

```haskell
data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val i) = i
eval (Add x y) = eval x + eval y
```

If we extend this language with variables, we need to extend the evaluator with an environment, and deal with possible failure if a variable is undefined (we omit the Val and Add cases):

```haskell
data Expr = Val Int | Add Expr Expr | Var String

type Env = [(String, Int)]

eval :: Expr -> ReaderT Env Maybe Int
eval (Var x) = do env <- lift ask
                  lift (lookup x env)

Here, the Maybe monad captures possible failure, and is transformed into a reader monad using the ReaderT transformer to store the environment, which is retrieved using ask. The lift operation allows functions in the inner Maybe monad to be called. We can extend the language further, with random number generation:

```haskell
data Expr = Val Int | Add Expr Expr | Var String | Random Int

eval :: Expr -> RandT (ReaderT Env Maybe) Int
eval (Var x) = do env <- lift ask
                 lift (lift (lookup x env))

We have added another transformer to the stack, RandT, and added lift where necessary to access the appropriate monads in the stack. We have built this interpreter from reusable components, the Maybe monad and ReaderT and RandT transformers, which is clearly a good thing. A problem, however, is that the use of lift is a little noisy, and will only get worse if we add more monads to the stack, such as IO. Bigger problems occur if we need to permute the order of the transformers, or invoke a function which uses a subset:

```haskell
permute :: ReaderT Env (RandT Maybe) a ->
         RandT (ReaderT Env Maybe) a

dropReader :: RandT Maybe a ->
            RandT (ReaderT Env Maybe) a
```

These problems mean that, in general, there is little motivation for separating effects, and a temptation to build an application around
one general purpose monad capturing all of an application’s state and exception handling needs. This is undesirable, as it limits reuse and hinders modular design. It would be desirable instead to be able to separate effects into specific modular components such as console I/O, file and network handling and operating system interaction, for the same reason that it is desirable to separate the pure fragments of a program from the impure fragments using the IO monad. That is, the program’s type would give more precise information about its behaviour, making the program easier to reason about and reducing the possibility of errors.

In this paper, I present an alternative approach to combining effects in a pure functional programming language, based on handlers of algebraic effects [3], and implemented directly as a domain specific language embedded in a dependently typed host, Idris [4, 5].

1 Contributions

This paper takes as its hypothesis that algebraic effects provide a cleaner, more composable and more flexible notation for programming with side effects than monad transformers. Although they are not equivalent in power (for example, unlike algebraic effects, monads can express continuations) many common side-effects are captured. The main contribution of this paper is a notation for describing and combining side effects using Idris. More specifically:

• An Embedded Domain Specific Language (DSL), Effects, for capturing algebraic effects, in such a way that they are easily composable, and translatable to a variety of underlying contexts using effect handlers.

• A collection of example effects (State, Exceptions, File and Console I/O, random number generation and non-determinism) and their handlers. I show how alternative handlers can be used to evaluate effects in different contexts, for example using an alternative handler to run interactive programs in a pure context.

• I give example programs which combine effects, including an interpreter for an imperative language with mutable variables, to illustrate how effectful applications may be structured.

The Effects DSL makes essential use of dependent types, firstly to verify that a specific effect is available to an effectful program using simple automated theorem proving, and secondly to track the state of a resource by updating its type during program execution. In this way, we can use the Effects DSL to verify implementations of resource usage protocols.

I describe how to use Effects in Section 2, how it is implemented in Section 3, and give a larger example in Section 4. It is distributed with Idris\(^1\). All of the examples in this paper are available online at http://idris-lang.org/effects.

2. Effects: an Embedded DSL for Effects Management

In this section, I introduce Effects, an embedded domain specific language for managing computational effects in Idris. I will introduce specific features of Idris as required. In particular, we will use implicit conversions and default implicit arguments in the implementation of Effects. A full tutorial is available elsewhere [5]. First, I describe how to use effects which are already defined in the language in order to implement the evaluator described in the introduction. Then, I show how new effects may be implemented.

The framework consists of a DSL representation Eff for combining effects, EffM for combining mutable effects, and implementations of several predefined effects. We refer to the whole framework with the name Effects.

2.1 Programming with Effects

Programs in the Effects language are described using the following data type, in the simplest case (I will defer introducing the constructors until later):

\[
\text{Eff : (m : Type -> Type) -> (es : List EFFECT) -> (a : Type) -> Type}
\]

Note that function types in Idris take the form \((x : a) -> b\), with an optional name \(x\) on the domain of the function. This is primarily to allow the name \(x\) to be used in the codomain, although it is also useful for documenting the purpose of an argument.

Eff is parameterised over a computation context, \(m\), which describes the context in which the effectful program will be run, a list of side effects \(es\) that the program is permitted to use, and the program’s return type \(a\). The name \(m\) for the computation context is suggestive of a monad, but there is no requirement for it to be so.

Side effects are described using the EFFECT type; we will refer to these as concrete effects. For example:

\[
\text{STATE : Type -> EFFECT}
\]

\[
\text{EXCEPTION : Type -> EFFECT}
\]

\[
\text{FILEIO : Type -> EFFECT}
\]

\[
\text{STDIO : EFFECT}
\]

\[
\text{RND : EFFECT}
\]

States are parameterised by the type of the state being carried, and exceptions are parameterised by a type representing errors. File I/O allows a single file to be processed, with the type giving the current state of the file (i.e. closed, open for reading, or open for writing). Finally, STDIO and RND permit console I/O and random number generation respectively. For example, a program with some integer state, which performs console I/O and which could throw an exception of type Error might have the following type:

\[
\text{example : Eff IO [EXCEPTION Error, STDIO, STATE Int] ()}
\]

More generally, a program might modify the set of effects available, for example when adding a new effect or updating a dependently typed state. In this case, we describe programs using EffM:

\[
\text{EffM : (m : Type -> Type) -> (es : List EFFECT) -> (a : Type) -> Type}
\]

EffM is parameterised over the context and type as before, but separates input effects \(es\) from output effects \(es'\). In fact, Eff is defined in terms of EffM, with equal input/output effects.

We adopt the convention that the names \(es\) and \(fs\) refer to a list of input effects, and \(es'\) and \(fs'\) refer to a list of output effects.

2.1.1 First example: State

In general, an effectful program implemented in the EffM structure has the look and feel of a monadic program in Haskell, since EffM supports do-notation. To illustrate basic usage, let us implement a stateful function, which tags each node in a binary tree with a unique integer, depth first, left to right. We declare trees as follows:

\[
\text{data Tree a = Leaf} \mid \text{Node (Tree a) a (Tree a)}
\]

To tag each node in the tree, we write an effectful program which, for each node, tags the left subtree, reads and updates the state, tags the right subtree, then returns a new node with its value tagged. The type expresses that the program requires an integer state:

\[
\text{tag : Tree a -> Eff m [STATE Int] (Tree (Int, a))}
\]

The implementation traverses the tree, using get and put to manipulate state:

\[^1\text{http://idris-lang.org/}\]
tag Leaf = return Leaf

Each effect is associated with a resource. For example, the resource associated with STATE Int is the integer state itself. To run an effectful program, we must create a list of resources corresponding to the list of supported effects, and instantiate m. Here we instantiate m with id, resulting in a pure function.

tagFrom : Int -> Tree a -> Tree (Int, a)
tagFrom x t = runPure [x] (tag t)

In general, to run an effectful program, we use one of the functions run, runWith or runPure, instantiating an environment with resources corresponding to each effect:

run : Applicative m =>
  Env m es -> EffM m es es' a -> m a
runWith : (a -> m a) ->
  Env m es -> EffM m es es' a -> m a
runPure : Env id es -> EffM id es es' a -> a

Corresponding functions runEnv, runWithEnv and runPureEnv also return the final resources. The reason run needs m to be an applicative functor is that it uses pure to inject a pure value into m. If this is inconvenient, runWith can be used. Note that there is no requirement that m is a monad; any type transformer is fine. In particular, id can be used to translate an effectful program into a pure function. As we will see, the particular choice of m can be important. Programs with exceptions, for example, can be run in the context of IO, Maybe or Either. We will return to the definition of Env in Section 3.2. For now, it suffices to know that it is a heterogeneous list of values for the initial resources es.

2.1.2 Labelled Effects

When we invoke an effect, Effects internally searches the list of available effects to check if it is supported, and invokes the effect using the corresponding resource. This leads to an important question: what if we have more than one effect which supports the function? A particular situation where this arises is when we have more than one integer state. Then, we use these labels to disambiguate the states. To increment the count at each leaf, we use update, which combines a get and a put by applying a function to the state:

tagCount : Tree a -> Eff m [Tag := STATE Int, Count := STATE Int] (Tree (Int, a))

tagCount Leaf
  = do Count := update (+1)
  return Leaf
tagCount (Node l x r) = do l' <- tagCount l
lbl <- get
Tag := put (lbl + 1)
r' <- tagCount r
return (Node l' (lbl, x) r')

In order to retrieve the count afterwards, we will need access to the environment after running tagCount. We use runPureEnv, which returns the final resource states as well as the result:

runPureEnv : Env id xs ->
EffM id xs xs' a -> (Env id xs', a)

to initialise the environment, we give the label name along with the initial value of the resource (where := initialises a label). The order is important here, since resources are initialised in the order given by the list of supported effects.

runPureEnv [Tag := 0, Count := 0] (tagCount t)

And finally, to implement a pure wrapper function which returns a pair of the count of leaves and a labelled tree, we call runPureEnv with the initial resources, and match on the returned resources to retrieve the leaf count:

tagCountFrom : Int -> Tree a -> Tree (Int, a)
tagCountFrom x t
  = let (_, Count := leaves), tree) = runPureEnv [Tag := 0, Count := 0] (tagCount t)
in (leaves, tree)

2.1.3 An Effectful Evaluator revisited

To implement the effectful evaluator from the introduction in Effects, we support exceptions, random numbers and an environment mapping from String to Int:

Vars : Type
Vars = List (String, Int)

The evaluator invokes supported effects where needed. We use the following effectful functions:

get : Eff m [STATE x] x
raise : a -> Eff m [EXCEPTION a] b
rndInt : Int -> Int -> Eff m [RND] Int

The evaluator itself is written as an instance of Eff:

eval : Expr ->
Eff m [EXCEPTION String, RND, STATE Vars] t

The implementation simply invokes the required effects with Effects checking that they are available:

eval (Val x) = return x
eval (Var x) = do vs <- get
  Nothing => raise (*Error " ++ x)
  Just val => return val

eval (Add l r) = [val] eval l + eval r []
eval (Random upper) = rndInt 0 upper
Remark: We have used idiom brackets [20] in this implementation, to give a more concise notation for applicative programming with effects. An application inside idiom brackets, [1 f a b c d ] translates directly to:

```
pure f <$> a <*> b <*> c <*> d
```

In order to run this evaluator, we must provide initial values for the resources associated with each effect. Exceptions require the unit resource, random number generation requires an initial seed, and the state requires an initial environment. We instantiate m with Maybe to be able to handle exceptions:

```
runEval : List (String, Int) -> Expr -> Maybe Int
runEval env expr = run [(), 123456, env] (eval expr)
```

Extending the evaluator with a new effect, such as STDIO is a matter of extending the list of available effects in its type. We could use this, for example, to print out the generated random numbers:

```
eval : Expr ->
    Eff m [ EXCEPTION String, STDIO, RND, STATE Vars ] t ...
    eval (Random upper) = do num <- rndInt 0 upper
                        putStrLn (show num)
                        return num
```

We can insert the STDIO effect anywhere in the list without difficulty. The only requirements are that its initial resource is in the corresponding position in the call to run, and that run instantiates a context which supports STDIO, such as IO:

```
runEval : List (String, Int) -> Expr -> IO Int
runEval env expr = run [(), (), 123456, env] (eval expr)
```

### 2.2 Implementing effects

In order to implement a new effect, we define a new type (of kind Effect) and explain how that effect is interpreted in some underlying context m. An Effect describes an effectful computation, parameterised by an input resource res, an output resource res', and the type of the computation t.

```
Effect : Type
Effect = (res : Type) -> (res' : Type) ->
    (t : Type) -> Type
```

We describe effects as algebraic data types. To run an effect, we require an interpretation in a computation context m. To achieve this, we make effects and contexts instances of a type class, Handler, which has a method handle explaining this interpretation:

```
class Handler (e : Effect) (m : Type -> Type)
    where handle : res -> (eff : e res res' t) ->
                  (res' -> t -> m a) -> m a
```

Type classes in Idris may be parameterised by anything. This means not only types, but also values, and even other type classes. Thus, if a parameter is anything other than a Type, it must be given a type label explicitly, like e and m here.

Handlers are parameterised by the effect they handle, and the context in which they handle the effect. This allows several different context-dependent handlers to be written, e.g. exceptions could be handled differently in an IO setting than in a Maybe setting. When effects are combined, as in the evaluator example, all effects must be handled in the context in which the program is run.

An effect e res res' t updates a resource type res to a resource type res', returning a value t. The handler, therefore, implements this update in a context m which may support side effects. The handler is written in continuation passing style. This is for two reasons: firstly, it returns two values, a new resource and the result of the computation, which is more cleanly managed in a continuation than by returning a tuple; secondly, and more importantly, it gives the handler the flexibility to invoke the continuation any number of times (zero or more).

An Effect, which is the internal algebraic description of an effect, is promoted into a concrete EFFECT, which is expected by the EffM structure, with the MkEff constructor:

```
data EFFECT : Type where
  MkEff : Type -> Effect -> EFFECT
```

MkEff additionally records the resource state of an effect. In the remainder of this section, we describe how several effects can be implemented in this way: mutable state; console IO; exceptions; files; random numbers, and non-determinism.

#### 2.2.1 State

In general, effects are described algebraically in terms of the operations they support. In the case of State, the supported effects are reading the state (Get) and writing the state (Put).

```
data State : Effect where
  Get : State a a a
  Put : b -> State a b ()
```

The resource associated with a state corresponds to the state itself. So, the Get operation leaves this state intact (with a resource type a on entry and exit) but the Put operation may update this state (with a resource type a on entry and b on exit). That is, a Put may update the type of the stored value. We can implement a handler for this effect, for all contexts m, as follows:

```
instance Handler State m where
  handle st Get k = k st st
  handle st (Put n) k = k n ()
```

When running Get, the handler passes the current state to the continuation as both the new resource and the result of the computation (the first argument of the continuation k) as well as the return value of the computation (the second argument of the continuation). When running Put, the new state is passed to the continuation as the new resource value.

We then convert the algebraic effect State to a concrete effect usable in an Effects program using the STATE function, to which we provide the initial state type as follows:

```
STATE : Type -> EFFECT
STATE t = MkEff t State
```

We adopt the convention that algebraic effects, of type Effect, have an initial upper case letter. Concrete effects, of type EFFECT, are correspondingly in all upper case.

Algebraic effects are promoted to Effects programs with concrete effects by using the mkEffect function. We will postpone giving the type of mkEffect until Section 3.1.2. For now, it suffices to know that it converts an Effect to an effectful program. To create the get and put functions, for example:

```
get : Eff m [STATE x] x
get = mkEffect Get
put : x -> Eff m [STATE x] ()
put val = mkEffect (Put val)
```

We may also find it useful to mutate the type of a state, considering that states may themselves have dependent types (we may, for example, add an element to a vector in a state). The Put constructor supports this, so we can implement putM to update the state's type:

```
putM : y -> EffM m [STATE x] [STATE y] ()
putM val = mkEffect (Put val)
```
Finally, it may be useful to combine `get` and `put` in a single update:

\[
\text{update} : (x \rightarrow x) \rightarrow \text{Eff m} \ [\text{STATE} \ x] ()
\]

\[
\text{update f = do val <- get; put (f val)}
\]

### 2.2.2 Console I/O

Consider a simplified version of console I/O which supports reading and writing strings. There is no associated resource, although in an alternative implementation we may associate it with an abstract world state, or a pair of handles for an alternative implementation we may associate it with an abstract type, and since raising an exception causes computation to abort, raising an exception can return a value of any type:

\[
\text{update f = do val <- get; put (f val)}
\]

#### 2.2.3 Exceptions

The exception effect supports only one operation, `Raise`. Exceptions are parameterised over an error type `e`, so `Raise` takes a single argument to represent the error. The associated resource is of unit type, and since raising an exception causes computation to abort, raising an exception can return a value of any type:

\[
\text{update f = do val <- get; put (f val)}
\]
Alternative handlers could use a different, possibly more secure approach. In any case, we can implement a function which returns a random number between a lower and upper bound as follows:

```
rdnInt : Int -> Int -> Eff m [RND] Int
rdnInt lower upper = do v <- mkEffect GetRandom
                    return (v 'mod' (upper - lower) + lower)
```

### 2.2.5 Resource management: Files

Instead of implementing all I/O operations as a single effect, as with the IO monad, we can separate operations into more fine-grained effects. Another I/O related effect which we can handle separately is file management. Here, we can take advantage of the resource associated with an effect, and the fact that resource types are mutable, by associating the effect with a file handle parameterised by its current state (i.e. closed, open for reading, or open for writing). File handles are represented as follows, where File is a primitive file handle provided by the IDRIS library:

```haskell
data FileIO : Effect where
  Open : FilePath -> (m : Mode) -> FileIO m
  Close : FileIO (OpenFile m) ()
  ReadLine : FileIO (OpenFile Read) (OpenFile Read) String
  WriteLine : String -> FileIO (OpenFile Write)
  EOF : FileIO (OpenFile Read) Bool
```

We can see from this declaration that opening a file moves from an empty resource to a file open for a specific purpose and that closing a file removes the file resource. Only files which are open for reading may be read, or tested for end of file, and only files open for writing may be written to. Any violation of this resource access protocol will result in a type error. In general, we can use the effect signature of a function to manage resources safely, subsuming the resource management DSL we have previously implemented [9].

The FILE_IO effect is parameterised over the current state of a file resource with which it is associated:

```haskell
FILE_IO : Type -> EFFECT
```

The type of open expresses that the resource changes from a unit to an open file:

```haskell
open : FilePath -> (m : Mode) ->
      EffM IO [FILE_IO ()] [FILE_IO (OpenFile m)] ()
```

Note that opening a file may fail. We will deal with such exceptional behaviour shortly. Using EffM, we have expressed that opening a file causes a change in the resource state. It is then only possible to close a file if there is an open file available:

```haskell
close : EffM IO [FILE_IO (OpenFile m)] [FILE_IO ()] ()
```

Reading is only possible from a file handle opened for reading:

```haskell
readLine : Eff IO [FILE_IO (OpenFile Read)] String
```

As with STATE, we can use labelled resources if we require more than one file handle at a time. We have handlers for FileIO for the IO context, which does not handle exceptions (in which case failing to open a file is a fatal run-time error), and an IOExcept e context which is IO augmented with exceptions of type e and an instance of the Catchable class:

```haskell
instance Catchable IOExcept err a
instance IOExcept err (file :: FilePath) (open :: Mode) (IOExcept err a)
```

Assuming we are in a state where we have a file handle available and open for reading, we can read the contents of a file into a list of strings:

```haskell
readLines : Eff (IOExcept String)
          [FILE_IO (OpenFile Read)] (List String)
readLines = readAcc [] where
  readAcc : List String ->
           Eff (IOExcept String)
           [FILE_IO (OpenFile Read)] (List String)
  readAcc acc = do e <- eof
                  if (not e) then do str <- readLine
                                 readAcc (str :: acc) else return acc
```

To read a file, given a file path, into a list of strings, reporting an error where necessary, we can write the following readFile function. We add STDIO to the list of effects so that we can report an error to the console if opening the file fails:

```haskell
readFile : FilePath -> Eff (IOExcept String)
          [FILE_IO (), STDIO] (List String)
readFile path = catch (do open path Read
                          lines <- readLines
                          close
                          return lines)
               (err => do putStrLn ("Failed " ++ err)
                      return [])
```

The effect type of readFile means that we must begin and end with no open file. This means that omitting the close would result in a compile time type error. It would also be a type error to try to invoke readLines before the file was open, or if the file was opened for writing instead.

### 2.2.6 Non-determinism

Following [3], non-determinism can be implemented as an effect Selection, in which a Select operation chooses one value non-deterministically from a list of possible values:

```haskell
data Selection : Effect where
  Select : List a -> Selection () () a
```

We can handle this effect in a Maybe context, trying every choice in a list given to Select until the computation succeeds:

```haskell
instance Handler Selection Maybe where
  handle _ (Select xs) k = tryAll xs where
    tryAll [] = Nothing
    tryAll (a :: xs) = case k () x of
      Nothing => tryAll xs
      Just v => Just v
```

The handler for Maybe produces at most one result, effectively performing a depth first search of the values passed to Select. The handler runs the continuation for every element of the list until the result of running the continuation succeeds. Alternatively, we can find every possible result by handling selection in a List context:

```haskell
instance Handler Selection List where
  handle r (Select xs) k = concatMap (k r) xs
```

We can use the Selection effect to implement search problems by non-deterministically choosing from a list of candidate solutions.
For example, a solution to the n-Queens problem can be implemented as follows. First, we write a function which checks whether a point on a chess board attacks another if occupied by a Queen:

\[ no\_attack (x, y) (x', y') = x \neq x' \&\& y \neq y' \&\& \text{abs} (x - x') \neq \text{abs} (y - y) \]

Then, given a column and a list of Queen positions, we find the rows on which a Queen may safely be placed in that column:

\[ \text{rowsIn} : \text{Int} \rightarrow \text{List} (\text{Int, Int}) \rightarrow \text{List} \text{Int} \]

\[ \text{rowsIn} \text{col} \text{qs} = [ x | x \leftarrow [1..8], \text{all} (\text{no\_attack} (x, \text{col})) \text{qs} ] \]

Finally, we compute a solution by accumulating a set of Queen positions, column by column, non-deterministically choosing a position for a Queen in each column:

\[ \text{addQueens} : \text{Int} \rightarrow \text{List} (\text{Int, Int}) \rightarrow \text{Eff} \text{m} [\text{SELECT}] \text{List} (\text{Int, Int}) \]

\[ \text{addQueens} 0 \text{ qs} = \text{return} \text{ qs} \]

\[ \text{addQueens} \text{ col} \text{ qs} = \text{do} \text{ row} \leftarrow (\text{rowsIn} \text{ col} \text{ qs}) \]

\[ \text{addQueens} \text{col} \text{ qs} = (\text{addQueens} \text{col} - 1) ((\text{row}, \text{col}) :: \text{qs}) \]

We can run this in Maybe context, to retrieve one solution, or in List context, to retrieve all solutions. In a Maybe context, for example, we can define:

\[ \text{getQueens} : \text{Maybe} \text{ List} (\text{Int, Int}) \]

\[ \text{getQueens} = \text{return} [] \]

Then to find the first solution, we run getQueens at the REPL:

\[ \text{*>Queens*} \text{ show} \text{getQueens} \]

"Just [(4, 1), (2, 2), (7, 3), (3, 4), (6, 5), (8, 6), (5, 7), (1, 8)]" : String

Remark: It is important to note that when combining SELECT with other effects, the values of other resources are reset at the beginning of each select branch. This means, at least in the current implementation, that state cannot be shared between branches. While perhaps not so important for selection, this may be a problem for other control effects such as co-operative multithreading, for which we may need a more flexible handler if we wish to deal with shared state. We will deal with this issue in future work.

2.3 Effect Polymorphism

Since Effects is implemented as an embedding in a host language, we can exploit features of the host language. The means that we can write higher order functions, and functions which are polymorphic in their effects. For example, a mapEff function can be implemented corresponding to \text{map} for functors:

\[ \text{mapEff} : \text{Applicative} \text{m} \rightarrow (a \rightarrow \text{Eff} \text{m} \text{xs} \text{b}) \rightarrow \text{List} a \rightarrow \text{Eff} \text{m} \text{xs} \text{ (List b)} \]

\[ \text{mapEff} f [] = \text{pure} [] \]

\[ \text{mapEff} f (x :: xs) = [ f x :: \text{mapEff} f xs |] \]

This applies an effectful function across a list, provided that we are in an applicative context, and that the effectful function supports the correct set of effects.

3. The Effects DSL implementation

The implementation of Effects follows a common pattern in dependently typed DSL implementation, that of the well-typed interpreter [2, 6, 25]. We describe the DSL as a data type, \text{EffM}, which expresses the properties a program must satisfy in its type, with a corresponding interpreter. The type system guarantees that only programs which satisfy these properties can be interpreted.

3.1 Language representation

The \text{EffM} data type represents the \text{Effects} language constructs. It is parameterised over its computation context \text{m}, and indexed by the list of effects on input and the list of effects on output, as well as the return type of the computation:

\[ \text{data EffM} : (\text{m} : \text{Type} \rightarrow \text{Type}) \rightarrow (\text{es} : \text{List EFFECT}) \rightarrow (\text{es}' : \text{List EFFECT}) \rightarrow (a : \text{Type}) \rightarrow \text{Type} \]

For the common case of programs in which the input effects are the same as the output effects, we define \text{Eff}:

\[ \text{Eff} : (\text{m} : \text{Type} \rightarrow \text{Type}) \rightarrow (\text{es} : \text{List EFFECT}) \rightarrow (a : \text{Type}) \rightarrow \text{Type} \]

Eff \text{m es t} = \text{EffM m es es t}

The complete syntax is given in Figure 1 for reference. In this section, we describe the constructs in detail.

Remark: One way of looking at \text{EffM} is as a flexible monad transformer, augmenting an underlying monad with additional features. In this way we can combine algebraic effects with concepts more difficult to capture, such as partiality and continuations.

3.1.1 Basic constructs

In the simplest case, we would like to inject pure values into the \text{EffM} representation:

\[ \text{return} : a \rightarrow \text{EffM m es es a} \]

Additionally, we have \text{>>(=)} to support do notation:

\[ (\triangleright\triangleright) : \text{EffM m es es' a} \rightarrow (a \rightarrow \text{EffM m es' es'' b}) \rightarrow \text{EffM m es es'' b} \]

The type of \text{>>(=)} expresses that available effects may change, transitively, across a program. The output of the first action, \text{es'} becomes the input to the second.

Remark: \text{IDRIS} allows ad-hoc name overloading, as well as overloading via type classes. Since it does not support type inference at the top level, as full type inference is undecidable for dependently typed languages, such overloading can be resolved in a type directed way. It is important here, since we cannot make \text{EffM} an instance of \text{Monad}, but we would like to use do-notations as a natural way of implementing imperative code.

3.1.2 Invoking effects

Recall that an algebraic effect is promoted to an \text{Effects} program with the \text{mkEffect} function, the type of which has so far been postponed. We can invoke effects using a helper, \text{mkEffectP}, which requires a \text{proof}, \text{pf}, that the effect is available:

\[ \text{mkEffectP} : (\text{pf} : \text{EffElem e a es}) \rightarrow (\text{eff} : e a b t) \rightarrow \text{EffM m es (updateResTy es pf eff t)} \]

If we are invoking an effect \text{eff}, provided by the algebraic effect \text{e}, with an input resource type \text{a}, this is valid only if we can provide a proof that the algebraic effect is present in the input set of resources \text{es}. This concept is captured in the \text{EffElem} predicate:

\[ \text{data EffElem} : (\text{Type} \rightarrow \text{Type} \rightarrow \text{Type} \rightarrow \text{Type}) \rightarrow (\text{Type} \rightarrow \text{List EFFECT} \rightarrow \text{Type where} \]

\[ \text{Here} : \text{EffElem x a (MkEff a x :: es)} \]

\[ \text{There} : \text{EffElem x a es} \rightarrow \text{EffElem x a (y :: es)} \]

This proof serves two purposes: it provides a guarantee that the effect is available with the correct resource type; and, as we shall see shortly, it serves as an index into the resource set when we \text{evaluate} programs. After the effect has been handled, its resource type may have been updated. We must therefore update the output
resource state in the type. Again, we use the structure of the proof, to update the resource type from a to b.

```
data EffM : (m : Type -> Type) -> List EFFECT -> List EFFECT -> Type -> Type where
    return : a -> EffM m es es a
    (>>=) : EffM m es es' a -> (a -> EffM m es' es'' b) -> EffM m es es'' b
mkEffectP : (prf : EffElem e a es) -> (eff : e a b t) -> EffM m es (updateResTy es prf eff) t
liftP : (prf : SubList fs es) -> EffM m fs fs' t -> EffM m es (updateWith fs' es prf) t
(:-) : (l : ty) -> EffM m [x] [y] t -> EffM m [l :: x] [l :: y] t
new : Handler e m => res -> EffM m (MkEff res e :: es) (MkEff res' e :: es') a -> EffM m es es' a
catch : Catchable m err => EffM m es (updateResTy es prf eff) t
```

The problem with mkEffectP is that we must provide an explicit proof that the effect is available. Since the effect list is statically known, IDRIS ought to be able to find such a proof automatically.

IDRIS currently has limited proof search, based on reflection and reification of programs, but it is sufficient for constructing the required proofs. The reflection mechanism itself is beyond the scope of this paper. Briefly, we have the following function, which constructs a tactic for searching for proofs of EffElem:

```
findEffElem : Tactic
```

Then, we have a notation for giving default implicit arguments to functions. The mkEffect function is a wrapper for mkEffectP with a default argument which invokes the findEffElem tactic.

```
If the proof search fails, this causes a compile time error, reporting that the effect is not available.

```

The sub-program may itself update the effects it uses, so again the proof serves two purposes: Firstly, to ensure the effects are indeed available; and secondly, to be able to update the effects in the outer program once the sub-program is complete, as follows:

```
updateWith : (fs' : List a) -> (es : List a) -> SubList fs es -> List a
updateWith (y :: fs) (x :: es) (Keep rest) = y :: updateWithWith fs es rest
updateWith fs (x :: es) (Drop rest) = x :: updateWithWith fs es rest
```

An alternative base case could state that [] is a sub-list of any list, assuming that remaining items are dropped.

The sub-program may itself update the effects it uses, so again the proof serves two purposes: Firstly, to ensure the effects are indeed available; and secondly, to be able to update the effects in the outer program once the sub-program is complete, as follows:

```
updateWith : (fs' : List a) -> (es : List a) -> SubList fs es -> List a
updateWith (y :: fs) (x :: es) (Keep rest) = y :: updateWithWith fs es rest
updateWith fs (x :: es) (Drop rest) = x :: updateWithWith fs es rest
```

Again, we can construct the necessary proofs of SubList automatically, if the sub-program uses a valid set of effects, because all effects are statically known, using a reflected tactic findSubList. We implement a wrapper lift which builds this proof implicitly:

```
implicit
lift : {default tactics { reflect findSubList; } prf : SubList fs es} ->
      prog : EffM m es fs' t ->
      EffM m es (updateWithWith fs' es prf) t
```

The implicit modifier before lift states that this function can be used as an implicit conversion. Implicit conversions are inspired by a similar feature in Scala [24]. The effect of the implicit modifier is, intuitively, that lift will be applied to a program in EffM if it is required for type correctness. Since type checking is type directed it always has access to the required type of an expression, and the implicit coercions which produce that type, so applying conversions is simple.

Such conversions are, deliberately, limited. They cannot be chained, unlike implicit coercions in Coq, to avoid coherence problems. Furthermore, to avoid ambiguity problems, if there is more than one implicit conversion available then neither will be applied.

In the Effects library, only lift is implicit.

Remark: Using SubList as it is, rather than some notion of subset, means that in sub-programs the effects must appear in the same order as they appear in the caller’s effect list. This is not an inherent limitation; with improved proof search, we should also be able to support effect sets which are permutations of another. For the moment, to handle the (uncommon) case where permutations are needed, we provide a permute operation, where Perm es fs is a proof that es is a permutation of fs, though we omit further details from the present paper.

```
permute : Perm es fs -> Eff m fs t -> Eff m es t
```

3.1.4 Labelling effects

If we have an effectful program p with a single effect, we can label that effect using the (:->) operator:

```
(->) : (l : ty) ->
     EffM m [x] [y] t ->
     EffM m [l :: x] [l :: y] t
```
It is sufficient to handle single effects here, because labels can only apply to one effect at a time, and such effectful sub-programs can easily be invoked using an implicit `lift` as described above. Labelling effects does nothing more than adding a label to the effect and its corresponding resource.

### 3.1.5 Introducing effects

We can introduce a new effect in the course of an effectful program, provided that the effect can be handled in the current computation context `m:

```haskell
new : Handler e m =>
    res -> EffM m (MkEff res e :: es)
    (MkEff res' e :: es') a ->
    EffM m es es' a
```

This extends the list of effects, initialised with a resource of type `res`. Once the sub-program is complete, the resource for the new effect is discarded, as is clear from the type of `new`. The effect itself, `e`, is never given explicitly here, which means that it must be clear from the sub-program what the effect is. Typically, this means that the sub-program will be a function with an explicit type.

### 3.1.6 Handling failure

Finally, if the computation context `m` supports failure handling, we can use the catch construct to handle errors:

```haskell
catch : Catchable m err =>
    (prog : EffM m es es' a) ->
    (handler : err -> EffM m es es' a) ->
    EffM m es es' a
```

If the sub-program `prog` fails with an error, of type `err`, then the handler is called with the error. Note that both `prog` and `handler` transform the effect list from `es` to `es'`. If `prog` fails, then the resources are reset to their starting state. This requires some care, if the effect refers to external resources such as file handles.

### 3.2 The Effects interpreter

Running an effectful program, of type `EffM m es es' t`, should yield a computation of type `m t`, that is, the program returns a value of type `t` in some computation context `m`. We can interpret programs through a traversal of the `EffM` syntax, subject to:

- Keeping track of resources corresponding to each effect.
- Invoking the appropriate handlers whenever necessary.
- Returning two values as part of the computation: the result `t` and an updated collection resources.

To keep track of resources, we build an environment as a heterogeneous list to store the resource corresponding to each effect:

```haskell
data Env : (m : Type -> Type) ->
    List EFFECT -> Type where
Nil : Env m Nil
(::__) : Handler eff m =>
    a -> Env m es -> Env m (MkEff a eff :: es)
```

Using the (overloaded) `Nil` and `(__)` gives us access to list syntax for environments. They are parameterised over a computation context `m`, which allows an effect handler instance to be associated with each entry. As a result `EffM` programs can be independent of context, thus interpretable in several contexts, and effects and hence their handlers can change during execution.

Since we need to return two values, a result and an updated resource collection, we implement the evaluator in continuation passing style, with the two values passed to the continuation. This also helps when invoking handlers, which also require a continuation. The interpreter has the following type:

```haskell
effInt : Env m es -> EffM m es es' a ->
    (Env m es' -> a -> m b) -> m b
```

Effectful programs are invoked using the following form, calling `effInt` with a continuation which simply discards the environment when evaluation is complete.

```haskell
run : Applicative m =>
    Env m es -> EffM m es es' a a -> m a
run env prog = effInt env prog (\env, r => pure r)
runPure : Env id es -> EffM id es es' a a -> a
runPure env prog = effInt env prog (\env, r => r)
```

The full implementation of `effInt` is given for reference in Figure 2. The interpreter uses a number of helpers in order to manage effects. Firstly, to invoke a handler given a proof that an effect is available, we use `execEff`, defined as follows:

```haskell
execEff : Env m es -> (p : EffElem e res es) ->
    (eff : e res b a) ->
    (Env m (updateResTy es p eff) a -> a -> m t) ->
    m t
effInt env (mkEffectP prf effP) k
    = execEff env prf effP k
```

Invoking a sub-program with a smaller collection of effects involves dropping the unnecessary resources from the environment, invoking the sub-program, then rebuilding the environment reinstating the resources which were not used. We drop resources from an environment according to the `SubList` proof with `dropEnv`:

```haskell
dropEnv : Env m fs -> SubList es fs -> Env m es
dropEnv [] SubNil = []
dropEnv (v :: vs) (Keep rest) = v :: dropEnv vs rest
dropEnv (v :: vs) (Drop rest) = dropEnv vs rest
```

Correspondingly, `rebuildEnv` rebuilds the outer environment, updating the resources which were updated by the sub-programs:

```haskell
rebuildEnv : Env m fs' -> (prf : SubList fs es) ->
    Env m es -> Env m (updateWith fs' es prf)
rebuildEnv [] SubNil env = env
rebuildEnv (f :: fs) (Keep rest) (e :: env) = f :: rebuildEnv fs rest env
rebuildEnv fs (Drop rest) (e :: env) = e :: rebuildEnv es rest env
```

We can use `dropEnv` and `rebuildEnv` to interpret the execution of sub-programs, dropping resources before invoking, then rebuilding the environment before invoking the continuation:

```haskell
effInt (liftP prf effP) k
    = let env' = dropEnv env prf in
effInt env' effP (\envk, p' =>
    k (rebuildEnv envk prf envk) p')
```

To introduce a new effect using `new`, we simply extend the environment with the new resources before invoking the sub-program, and drop the extra resource before invoking the continuation:

```haskell
effInt env (new r prog) k
    = let env' = r :: env in
effInt env' prog (\(v :: envk), p' => k envk p')
Interpreting a catch involves using the catch method of the Catchable class. We rely on type directed overloading here to disambiguate the catch:

\[
\text{effInt env (catch prog handler) } k = \text{catch (effInt env prog k)} \quad (\\wedge e \Rightarrow \text{effInt env (handler e)} ) \quad k
\]

Finally, to interpret a labelled effect, we remove the label, interpret the resulting effect as normal, then replace the label (unlabel and label have the obvious definitions, with unlabel removing a label from a singleton environment, and relabel replacing it):

\[
\text{effInt env (l := prog) } k = \text{let env'} = \text{unlabel env in} \quad \text{effInt env'} \text{ prog \{envk, p' } \Rightarrow \text{k (relabel l envk) p'}}
\]

**4. Example: An Imperative Language Interpreter**

As a larger example, I present an interpreter for a small imperative language, Imp, which supports variables (requiring an updatable environment), and printing (requiring console I/O). We follow the well-typed interpreter pattern, using a context membership proof to guarantee that local variables are well-scoped. We separate expressions (Expr) from statements in the imperative language (Imp). First, we define the types, supporting integers, booleans and unit, and a function interpTy to convert these to IDRIS types:

\[
data Ty = TyInt \mid TyBool \mid TyUnit
\]

\[
interpTy : Ty \rightarrow Type
\]

Expressions include variables, random numbers and binary operators derived from IDRIS functions, and are indexed by a context G (of type Vect Ty n), and the type of the expression:

\[
data Expr = Vect Ty n \rightarrow Ty \rightarrow Type where
\]

\[
Val : \text{interpTy a } \rightarrow \text{Expr G a}
\]

\[
Var : \text{HasType i G t } \rightarrow \text{Expr G t}
\]

\[
Rnd : \text{Int } \rightarrow \text{Expr G TyInt}
\]

\[
Op : \text{interpTy a } \rightarrow \text{interpTy b } \rightarrow \text{interpTy c } \rightarrow \text{Expr G a } \rightarrow \text{Expr G b } \rightarrow \text{Expr G c}
\]

For brevity, we omit the definition of HasType. It is sufficient to know that HasType i G t states that variable i (a de Bruijn index) in context G has type t. Values of variables are stored in a heterogeneous list corresponding to a vector of their types, with a lookup function to retrieve these values:

\[
data Vars = Vect Ty n \rightarrow Type where
\]

\[
Nil : Vars Nil
\]

\[
(\_ : : ) : \text{interpTy a } \rightarrow \text{Vars G } \rightarrow \text{Vars (a : : G)}
\]

\[
lookup : \text{HasType i G t } \rightarrow \text{Vars G } \rightarrow \text{interpTy t}
\]

We can write an evaluator for this simple expression language in Effects, using the random number effect, and an environment corresponding to the context G:

\[
\text{eval : Expr G t } \rightarrow \text{Eff m [RND, STATE (Vars G)] (interpTy t)}
\]

\[
eval (Val x) = \text{return x}
\]

\[
eval (Var i) = \text{do vars <- get; putM vars'; return (lookup i vars)}
\]

\[
eval (Rnd upper) = \text{rn4Int 0 upper}
\]

\[
eval (Op op x y) = \{ [op (eval x)] (eval y) \}
\]

Using dependent types, we have expressed a correspondence between the context G under which the expression and the variables are defined. The imperative fragment is also indexed over a context G and the type of a program. We use the unit type for statements (specifically Print) which do not have a value:

\[
data Imp = Vect Ty n \rightarrow Ty \rightarrow Type where
\]

\[
\text{Let : Expr G t } \rightarrow \text{Imp (t : : G) u } \rightarrow \text{Imp G u}
\]

\[
\text{Print : Expr G TyInt } \rightarrow \text{Imp G TyUnit}
\]

\[
\text{interp : Imp G t } \rightarrow \text{Eff IO [STDIO, RND, STATE (Vars G)] (interpTy t)}
\]

In order to interpret Let, which introduces a new variable with a given expression as its initial value, we must update the environment. Before evaluating the scope of the Let binding, the environment must be extended with an additional value, otherwise the recursive call will be ill-typed because the state effect must be carrying an environment of the correct length and types. Therefore, we evaluate the expression with eval, extend the environment with the result, evaluate the scope, then drop the value.

\[
\text{interp (Let e sc) } = \{ \begin{array} {ll}
  \text{do e' } \leftarrow \text{eval e; vars } \leftarrow \text{get; putM (e' : vars);}
  \\
  \text{res } \leftarrow \text{interp sc; (e' := vars') } \leftarrow \text{get; putM vars'; return res}
\end{array} \}
\]

Calling eval is fine here, because it uses a smaller set of effects than interp. Also, note that if we forget to drop the value before returning, this definition will be ill-typed because the type of interp requires that the environment is unchanged on exit.

Interpreting an assignment simply involves evaluating the expression to be stored in the variable, then updating the state, where updateVar v vars val updates the variable at position v in the environment vars with the new value val:

\[
\text{interp } (\_ := \text{prog}) = \text{let env'} = \text{unlabel env in}
\]

\[
\text{effInt env' \text{ prog \{envk, p' } \Rightarrow \text{k (relabel l envk) p'}}
\]

**Figure 2. The Effects DSL interpreter**
For Print, we simply evaluate the expression and display it, relying on the STDIO effect:

```hs
interp (Print x) = do e <- eval x
    putStrLn (show e)
```

Given some program, `prog : Imp [] TyUnit`, a main program would need to set up the initial resources, then call the interpreter:

```hs
main = run [(), 123456, []] (interp prog)
```

Though small, this example illustrates the design of a complete application with Effects: a main program sets up the required set of resources and invokes the top level effectful program. This, in turn, invokes effectful programs as necessary, which may use at most the resources available at the point where they are invoked.

5. Related Work

The work presented in this paper arose from dissatisfaction with the lack of composability of monad transformers, and a belief that a dependently typed language ought to handle side-effects more flexibly, in such a way that it would be possible to reason about side-effecting programs. The inspiration for using an algebraic representation was Bauer and Pretnar’s $\textbf{Eff}$ [3], a language based around handlers of algebraic effects. In our Effects system, we have found type classes to be a natural way of implementing handlers, in that they allow different effect handlers to be run in different contexts. Other languages aim to bring effects into their type system, such as Disciple [18], Frank\(^2\) and Koka\(^3\). These languages are built on well-studied theoretical foundations [13, 16, 27, 28], which we have also applied in this paper. Algebraic effects have also been applied in Haskell, in order to combine I/O effects [30], and more recently [15] with an effect handler library.

In our approach, we have seen that the Idris type system can express effects by embedding, without any need to extend the language or type system. We make no attempt to infer effect types, however, since our intention is to specify effects and check that they are used correctly.

While our original motivation was to avoid the need for monad transformers [17] in order to compose effects, there is a clear relationship. Indeed, Eff\(^4\) is in a sense a monad transformer itself, in that it may augment an underlying monad with additional effects. Therefore, we can expect it to be possible to combine effects and monad transformers, where necessary. The problem with modularity of monad transformers is well-known, and addressed to some extent [14], though this still does not allow easy reordering of transformers, or reducing the transformer stack. The Effects approach encourages a more fine-grained separation of effects, by making it easy to call functions which use a smaller set of effects.

Our approach, associating resources with each effect, leads to a natural way of expressing and verifying resource usage protocols, by updating the resource type. This is a problem previously tackled by others, using special purpose type systems [32] or Haskell extensions [19], and in my own previous work [7, 9] by creating a DSL for resource management, but these are less flexible than the present approach in that combining resources is difficult. A related concept is Typestate [1, 29], which allows states to be tracked in types, though again, we are able to implement this directly rather than by extending the language. Implementing and reasoning about state-dependent interactive programs has also been studied using type theory [12, 21].

To some extent, we can now support imperative programming with dependent types, such as supported by Xanadu [33] and Ynot [23]. Ynot in particular is an axiomatic extension to Coq which allows reasoning about imperative programs using Hoare Type Theory [22]. Preconditions and postconditions on operations can be expressed in Effects with input and output resource types. The difficulty in imperative programming with dependent types is that updating one value may affect the type of another, though in our interpreter example in Section 4 we have safely used a dependent type in a mutable state.

6. Conclusions

Idris is a new language, with a full dependent type system. This gives us an ideal opportunity to revisit old problems about how to handle effects in a pure functional language. While the conventional approach, based on monads and transformers, has proved successful in Haskell, in this paper we have investigated an alternative approach and found it to be a natural method for defining and using effects. By linking each effect with a resource we can even track the state of resources through program execution, and reason about resource usage protocols such as file management.

The Effects approach has a number of strengths, and some weaknesses which we hope to address in future work. The main strength is that many common effects such as state, I/O and exceptions can be combined without the need for monad transformers. Effects can be implemented independently, and combined without further effort. Lifting is automatic; sub-programs using a smaller set of effects can be called without any explicit lifting operation, so as well as being easy to combine effects, programs remain readable.

We have described a number of effects which are representative in this setting, and there are several others we have not described which are easy to define, such as parsing, logging, and bounded mutable arrays. Arrays, statically bounded with a dependent type, could even lead to optimisations since the effect system can guarantee that only one copy is needed, therefore the array could have a compact representation with no run-time bounds checking.

Another advantage is that it will be possible to have fine-grained separation of systems effects. We can be precise about needing network support, CGI, graphics, the operating system’s environment, etc, rather than including all of these effects in one I/O monad.

While using Effects, we are still able to use monads. Effects work with monads rather than against them, and indeed effectful programs are generally translated to monadic programs. As a result, concepts which need to be monadic (for example, continuations) can remain so, and still work with Effects.

Further Work

The Effects system is promising as a general approach to effectful programming, but we have only just begun. At present, we are developing more libraries using Effects to assess how well it works in practice as a programming tool, as well as how efficient it is for realistic programs. The most obvious weakness of Effects is that algebraic effects cannot capture all monads. This does not appear to be a serious problem, however, given that Effects is designed to interact with monads, rather than to replace them. More seriously, but less obviously, there is a small interpreter overhead since Eff\(^4\) is represented as an algebraic data type, with an associated interpreter. We have not yet investigated in depth how to deal with this overhead, or indeed if it is a serious problem in practice, but we expect that partial evaluation [8] or a finally tagless approach [10] would be sufficient.

\(^2\)https://personal.cis.strath.ac.uk/conor.mcbride/pub/Frank/

\(^3\)http://research.microsoft.com/en-us/projects/koka/
Another weakness which we hope to address is that mixing control and resource effects is a challenge. For example, we cannot currently thread state through all branches of a non-deterministic search. If we can address this, it may be possible to represent more sophisticated effects such as co-operative multithreading, or even partiality. One way to tackle this problem could be to introduce a new method of the Handler class which manages resources more precisely, with a default implementation calling handle.

An implementation detail which could be improved without affecting usage of the library is that effectful sub-programs require ordering of effects to be preserved. We address this with a permute operator, requiring a proof that the new effect list is a permutation of the old, but ideally generating these proofs will be automatic.

The Effects implementation is entirely within the Idris language, with no extensions needed. It takes advantage of dependent types and theorem proving in several small but important ways: heterogeneous lists of resources, proofs of list membership and sublist predicates, and parameterisation of resources. Since modern Haskell supports many of these features, this leads to an obvious question: what would it take to implement Effects as an embedded library in Haskell? An interesting topic for further study would be whether this approach to combining effects would be feasible in a more mainstream functional language.

Monad transformers have served us well, and are a good fit for the Haskell type system. However, type systems are moving on. Perhaps now is the time to look for something more effective.

Acknowledgments

This work was supported by the Scottish Informatics and Computer Science Alliance (SICSA). My thanks to the inhabitants of the #idris IRC channel on irc.freenode.net for their feedback, and to the anonymous referees for their helpful comments.

References