Programming and Reasoning with Algebraic Effects and Dependent Types

Edwin C. 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.

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 — data may be corrupted or untrusted. Pure functional programming languages such as Haskell [25] manage such problems using *monads* [30], and allow multiple effects to be combined using a stack of *monad transformers* [16].

Monad transformers are an effective tool for structuring larger Haskell applications. A simple 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, there are some difficulties with building applications in this way. Two of the most important are that the order in which transformers are applied matters (that is, transformers do not commute easily), and that it is difficult to invoke a function which uses a subset of the transformers in the stack. To illustrate these problem, consider an evaluator, in Haskell, for a simple expression language:

```
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):

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

type Env = [(String, Int)]

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

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:

```
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
                  val <- lift (lift (lookup x env))
                  return val
eval (Random x) = do val <- getRandomR (0, x)
                     return val
```

We have added another transformer to the stack, `RandT`, and added `lift` where necessary to access the appropriate monads in the stack. We have been able to build this interpreter from reusable components — the `Maybe` monad and `ReaderT` and `RandT` transformers — which is clearly a good thing. One small 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`, though implementations of `lift` can be automated [14]. Bigger problems occur if we need to permute the order of the transformers, or invoke a function which uses a subset, for example:

```
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. It would be desirable, on the other

hand, to separate effects into specific 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 what the program is supposed to do, 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].

1.1 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 — monads and monad transformers can express more concepts — many common effectful computations 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. In particular, we can use 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 distinctive 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`.

[1] `http://idris-lang.org/`

2.1 Programming with `Effects`

Programs in the `Effects` language are described using the following data type, in the simplest case:

```
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:

```
STATE     : Type -> EFFECT
EXCEPTION : Type -> EFFECT
FILEIO    : Type -> EFFECT
STDIO     : EFFECT
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 is an effect which 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). The `STDIO` effect permits console I/O, and `RND` permits random number generation. 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:

```
example : Eff IO [EXCEPTION Error, STDIO, STATE Int] ()
```

More generally, a program might modify the set of effects available. This might be desirable for several reasons, such as adding a new effect, or to update an index of a dependently typed state. In this case, we describe programs using the `EffM` data type:

```
EffM : (m   : Type -> Type) ->
       (es  : List EFFECT) ->
       (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:

```
data Tree a = Leaf
            | 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:

```
tag : Tree a -> Eff m [STATE Int] (Tree (Int, a))
```

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

```
tag Leaf = return Leaf
tag (Node l x r)
     = do l' <- tag l
          lbl <- get; put (lbl + 1)
          r' <- tag r
          return (Node l' (lbl, x) r')
```

The `Effects` system ensures, statically, that any effectful functions which are called (`get` and `put` here) require no more effects than are available. The types of these functions are:

```
get : Eff m [STATE x] x
put : x -> Eff m [STATE x] ()
```

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 initialise each resource 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` are also available for cases when the final resources are required. 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 instead. Unlike the monad transformer approach, there is no requirement that `m` is a monad. Any type transformer is fine — in particular, `id` can be used if the effectful program can be translated 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 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. For example, to count the number of `Leaf` nodes in a tree while tagging nodes, we will need two integer states:

```
tagCount : Tree a ->
     Eff m [STATE Int, STATE Int] (Tree (Int, a))
```

What should be the effect of `get` and `put` in this program? Which effect do they act on? In practice, the earlier effect is chosen. While clearly defined, this is unlikely to be the desired behaviour, so to avoid this problem, effects may also be *labelled* using the `:::` operator. A label can be of any type, and an effect can be converted into a labelled effect using the `:-` operator:

```
(:::) : lbl -> EFFECT -> EFFECT
(:-)  : (l : lbl) -> EffM m [x] [y] t ->
                     EffM m [l ::: x] [l ::: y] t
```

In order to implement `tagCount` now, first we define a type for the labels. We have one state variable representing the leaf count, and one representing the current tag:

```
data Vars = Count | Tag
```

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 <- Tag :- 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):

```
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 -> (Int, 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 these effects are available:

```
eval (Val x) = return x
eval (Var x) = do vs <- get
                  case lookup x vs of
                    Nothing => raise ("Error " ++ x)
                    Just val => return val
eval (Add l r) = [| eval l + eval r |]
eval (Random upper) = rndInt 0 upper
```

Remark: We have used idiom brackets [19] in this implementation, to give a more concise notation for applicative programming with

effects. An application inside idiom brackets, `[| 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
```

Effects are typically described as algebraic data types. To *run* an effect, they must be handled in some specific computation context `m`. We achieve this by making effects and contexts instances of a 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 — 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 I/O; 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 value (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:

```
update : (x -> x) -> Eff m [STATE x] ()
update f = do val <- get; put (f val)
```

2.2.2 Console I/O

We consider a simplified version of console I/O which supports reading and writing strings to and from the console. There is no associated resource, although in an alternative implementation we may associate it with an abstract world state, or a pair of file handles for `stdin`/`stdout`. Algebraically:

```
data StdIO : Effect where
     PutStr : String -> StdIO () () ()
     GetStr : StdIO () () String

STDIO : EFFECT
STDIO = MkEff () StdIO
```

The obvious way to handle `StdIO` is by translating via the `IO` monad, which is implemented straightforwardly as follows:

```
instance Handler StdIO IO where
    handle () (PutStr s) k = do putStr s; k () ()
    handle () GetStr     k = do x <- getLine; k () x
```

Unlike the `State` effect, for which the handler worked in *all* contexts, this handler only applies to effectful programs run in an `IO` context. We can implement alternative handlers, and indeed there is no reason that effectful programs in `StdIO` must be evaluated in a monadic context. For example, we can define I/O stream functions:

```
data IOStream a
   = MkStream (List String -> (a, List String))

instance Handler StdIO IOStream where
    ...
```

A handler for `StdIO` in `IOStream` context generates a function from a list of strings (the input text) to a value and the output text. We can build a pure function which simulates real console I/O:

```
mkStrFn : Env IOStream xs -> Eff IOStream xs a ->
          List String -> (a, List String)
mkStrFn {a} env p input = case mkStrFn' of
                              MkStream f => f input
  where injStream : a -> IOStream a
        injStream v = MkStream (\x => (v, []))
        mkStrFn' : IOStream a
        mkStrFn' = runWith injStream env p
```

To illustrate this, we write a simple console I/O program which runs in any context which has a handler for `StdIO`:

```
name : Handler StdIO e => Eff e [STDIO] ()
name = do putStr "Name? "
          n <- getStr
          putStrLn ("Hello " ++ show n)
```

Using `mkStrFn`, we can run this as a pure function which uses a list of strings as its input, and gives a list of strings as its output. We can evaluate this at the IDRIS prompt:

```
*name> show $ mkStrFn [()] name ["Edwin"]
((), ["Name?" , "Hello Edwin\n"])
```

This suggests that alternative, pure, handlers for console I/O, or any I/O effect, can be used for unit testing and reasoning about I/O programs without executing any real I/O.

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.

```
data Exception : Type -> Effect where
     Raise : e -> Exception e () () b

EXCEPTION : Type -> EFFECT
EXCEPTION e = MkEff () (Exception e)
```

The semantics of `Raise` is to abort computation, therefore handlers of exception effects do not call the continuation `k`. In any case, this should be impossible since passing the result to the continuation would require the ability to invent a value in any arbitrary type `b`! The simplest handler runs in a `Maybe` context:

```
instance Handler (Exception a) Maybe where
     handle _ (Raise err) k = Nothing
```

Exceptions can be handled in any context which supports some representation of failed computations. In an `Either e` context, for example, we can use `Left` to represent the error case:

```
instance Handler (Exception e) (Either e) where
     handle _ (Raise err) k = Left err
```

Given that we can raise exceptions in an `Effects` program, it is also useful to be able to catch them. The `catch` operation runs a possibly failing computation `comp` in some context `m`:

```
catch : Catchable m err =>
        (comp : EffM m xs xs' a) ->
        (handler : err -> EffM m xs xs' a) ->
        EffM m xs xs' a
```

Using `catch` requires that the computation context `m` supports throwing and catching of errors, using the `Catchable` class implemented as part of the IDRIS library:

```
class Catchable (m : Type -> Type) t where
    throw : t -> m a
    catch : m a -> (t -> m a) -> m a
```

This is implemented directly as part of the `Effects` DSL. There is a simple instance for `Maybe`, limited to the unit error:

```
instance Catchable Maybe () where
    throw () = Nothing
    catch Nothing  h = h ()
    catch (Just x) h = Just x
```

The `Either e` instance allows that any type be thrown as an error:

```
instance Catchable (Either e) e where
    throw x = Left x
    catch (Left err) h = h err
    catch (Right x)  h = (Right x)
```

2.2.4 Random numbers

Random number generation can be implemented as an effect, with the resource tracking the *seed* from which the next number will be generated. The `Random` effect supports one operation, `getRandom`, which requires an `Int` resource and returns the next number:

```
data Random : Type -> Type -> Type -> Type where
     GetRandom : Random Int Int Int

RND : EFFECT
RND = MkEff Int Random
```

Handling random number generation shows that it is a state effect in disguise, where the effect updates the seed. This is a simple linear congruential pseudo-random number generator:

```
instance Handler Random m where
    handle seed GetRandom k
         = let seed' = 1664525 * seed + 1013904223 in
               k seed' seed'
```

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:

```
rndInt : Int -> Int -> Eff m [RND] Int
rndInt 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. After Console I/O, 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 file handling effect with an individual file, 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:

```
data Mode = Read | Write

data OpenFile : Mode -> Type where
     FH : File -> OpenFile m
```

When we declare the `FileIO` algebraic effect type, we express in the resource transitions how each effect changes the state of the resource:

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

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

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

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

Note that opening a file may fail — we will deal with 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:

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

Reading is only possible from a file opened for reading:

```
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:

```
data IOExcept err a
ioe_lift : IO a -> IOExcept err a

instance Catchable IOExcept err
```

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:

```
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 (reverse 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:

```
readFile : String -> 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:

```
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:

```
instance Handler Selection Maybe where
     handle _ (Select xs) k = tryAll xs where
         tryAll [] = Nothing
         tryAll (x :: 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:

```
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 : (Int, Int) -> (Int, Int) -> Bool
no_attack (x, y) (x', y')
  = x /= x' && y /= y' && abs (x - x') /= 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 using a list comprehension:

```
rowsIn : Int -> List (Int, Int) -> List Int
rowsIn col qs
   = [ x | x <- [1..8], all (no_attack (x, col)) 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.

```
addQueens : Int -> List (Int, Int) ->
            Eff m [SELECT] (List (Int, Int))
addQueens 0   qs = return qs
addQueens col qs
   = do row <- select (rowsIn col qs)
        addQueens (col - 1) ((row, col) :: 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:

```
getQueens : Maybe (List (Int, Int))
getQueens = run [()] (addQueens 8 [])
```

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

```
*Queens> show 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 `fmap` for functors:

```
mapEff : Applicative m =>
         (a -> Eff m xs b) -> List a -> Eff m xs (List b)
mapEff f []        = pure []
mapEff f (x :: xs) = [| f x :: 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, 24]. We describe the DSL as a data type, `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 `EffM` data type represents the `Effects` language constructs. It is parameterised over its computation context `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:

```
data EffM : (m   : Type -> Type) ->
            (es  : List EFFECT) ->
            (es' : List EFFECT) -> (a : Type) -> Type
```

For the common case of programs in which the input effects are the same as the output effects, we define `Eff`:

```
Eff : (m : Type -> Type) ->
      (es : List EFFECT) -> (a : Type) -> Type
Eff m es t = 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 `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 `EffM` representation:

```
return : a -> EffM m es es a
```

Additionally, we have `(>>=)` to support `do` notation:

```
(>>=) : EffM m es es' a ->
        (a -> EffM m es' es'' b) -> EffM m es es'' b
```

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

Remark: 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 `EffM` an instance of `Monad`, but we would like to use `do`-notation as a natural way of implementing imperative code.

3.1.2 Invoking effects

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

```
mkEffectP : (prf : EffElem e a es) ->
            (eff : e a b t) ->
            EffM m es (updateResTy es prf eff) t
```

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

```
data EffElem : (Type -> Type -> Type -> Type) ->
               Type -> List EFFECT -> Type where
     Here : EffElem x a (MkEff a x :: es)
     There : EffElem x a es -> 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

```
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 es' a -> (err -> EffM m es es' a) -> EffM m es es' a
```

Figure 1. The `Effects` DSL data type

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`.

```
updateResTy : (es : List EFFECT) ->
              EffElem e a es -> e a b t -> List EFFECT
updateResTy {b} (MkEff a e :: es) Here n
    = (MkEff b e) :: es
updateResTy (x :: es) (There p) n
    = x :: updateResTy es p n
```

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.

```
mkEffect : {default tactics { reflect findEffElem; }
              prf : EffElem e a es} ->
           (eff : e a b t) ->
           EffM m es (updateResTy es prf eff) t
mkEffect {prf} e = mkEffectP prf e
```

Fortunately, there is no need for a user of the library to know anything about this proof search mechanism. Tactic construction for automated proof search is related to the `Ltac` [11] system in Coq, the intention being to make simple proofs automatable.

3.1.3 Effectful sub-programs

As well as invoking algebraic effects directly, we would like to be able to invoke sub-programs, which may use all of the effects available, or a subset. To achieve this, we use the `liftP` construct:

```
liftP : (prf : SubList fs es) ->
        (prog : EffM m fs fs' t) ->
        EffM m es (updateWith fs' es prf) t
```

This requires a proof that the effects available to the sub-program `prog` are a subset (strictly, a sub-list) of the effects available to the outer program, expressed using the following predicate:

```
data SubList : List a -> List a -> Type where
     SubNil : SubList [] []
     Keep   : SubList es fs ->
              SubList (x :: es) (x :: fs)
     Drop   : SubList es fs ->
              SubList es (x :: fs)
```

This predicate describes a sublist in terms of a larger list by saying, for each element of the larger list, whether it is kept or dropped. 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 :: updateWith fs es rest
updateWith fs        (x :: es) (Drop rest)
           = x :: updateWith fs es rest
updateWith []        []        SubNil      = []
```

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 fs fs' t) ->
       EffM m es (updateWith fs' fs prf) t
lift {prf} e = lift prf e
```

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 [23] — 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`:

```
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:

```
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, being passed 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 the state there were in at the start. 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 mostly through a simple traversal of the `EffM` syntax, subject to the considerations the we need to:

- Keep track of *resources* corresponding to each effect.
- Invoke the appropriate handlers where necessary.
- Return 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 heterogenous list to store the resource corresponding to each effect:

```
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 in the context. This is important both because it means `EffM` programs can be independent of context, thus interpretable in several contexts, and because effects and hence their handlers may 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:

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

This takes an input set of resources, and a program, and the continuation to run on the result and updated environment. Effectful programs are invoked using a function of the following form, calling `effInt` with a continuation which simply discards the environment when evaluation is complete.

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

runPure : Env id es -> EffM id es es' 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:

```
execEff : Env m es -> (p : EffElem e res es) ->
          (eff : e res b a) ->
          (Env m (updateResTy es p eff) -> a -> m t) ->
          m t
execEff (val :: env) Here eff' k
    = handle val eff' (\res, v => k (res :: env) v)
execEff (val :: env) (There p) eff k
    = execEff env p eff (\env', v => k (val :: env') v)
```

The proof locates the effect and handler in the environment. Following the interpreter and the handlers, `execEff` is written in continuation passing style, and is used directly to execute effects:

```
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`:

```
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:

```
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:

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

```
effInt : Env m es -> EffM m es es' a -> (Env m es' -> a -> m b) -> m b
effInt env (return x) k = k env x
effInt env (prog >>= c) k = effInt env prog (\env', p' => effInt env' (c p') k)
effInt env (mkEffectP prf effP) k = execEff env prf effP k
effInt env (liftP prf effP) k = let env' = dropEnv env prf in
                                    effInt env' effP (\envk, p' => k (rebuildEnv envk prf env) p')
effInt env (new r prog) k = let env' = r :: env in
                                effInt env' prog (\ (v :: envk), p' => k envk p')
effInt env (catch prog handler) k = catch (effInt env prog k)
                                          (\e => effInt env (handler e) k)
effInt env (l :- prog) k = let env' = unlabel env in
                                  effInt env' prog (\envk, p' => k (relabel l envk) p')
```

Figure 2. The `Effects` DSL interpreter

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:

```
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` in the `EffM` structure from the `catch` provided by the `Catchable` class:

```
effInt env (catch prog handler) k
     = catch (effInt env prog k)
             (\e => effInt env (handler e) 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):

```
effInt env (l :- prog) k
   = let env' = unlabel env in
         effInt env' prog (\envk, p' =>
                              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 | TyBool | TyUnit
interpTy : Ty -> Type
```

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

```
data Expr : Vect Ty n -> Ty -> Type where
     Val : interpTy a -> Expr G a
     Var : HasType i G t -> Expr G t
     Rnd : Int -> Expr G TyInt
     Op  : (interpTy a -> interpTy b -> interpTy c) ->
           Expr G a -> Expr G b -> 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 -> Type where
     Nil  : Vars Nil
     (::) : interpTy a -> Vars G -> Vars (a :: G)
lookup : HasType i G t -> Vars G -> 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`:

```
eval : Expr G t ->
       Eff m [RND, STATE (Vars G)] (interpTy t)
eval (Val x) = return x
eval (Var i) = do vars <- get
                  return (lookup i vars)
eval (Rnd upper) = rndInt 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 -> Ty -> Type where
     Let    : Expr G t -> Imp (t :: G) u -> Imp G u
     (:=)   : HasType i G t -> Expr G t -> Imp G t
     Print  : Expr G TyInt -> Imp G TyUnit
     (>>=)  : Imp G a ->
              (interpTy a -> Imp G b) -> Imp G b
     return : Expr G t -> Imp G t
```

Interpreting the imperative fragment requires the local variables to be stored as part of the state, as well as console I/O, for interpreting `Print`. We express this with the following type:

```
interp : Imp G t ->
     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 — 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 from the environment.

```
interp (Let e sc)
    = do e' <- eval e
         vars <- get; putM (e' :: vars);
         res <- interp sc
         (_ :: vars') <- get; putM vars'
         return res
```

Calling `eval` is fine here, because it uses a smaller set of effects than `interp`. Also, not 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'`:

```
interp (v := val)
     = do val' <- eval val
          update (\vars => updateVar v vars val')
          return val'
```

For `Print`, we simply evaluate the expression and display it, relying on the `STDIO` effect:

```
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:

```
main : IO ()
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 of effects was Bauer and Pretnar's *Eff* language [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 effect handlers, particular because they allow different effect handlers to be run in different contexts. Other languages aim to bring effects into their type system, such as Disciple [17], Frank[2] and Koka[3]. These languages are built on well-studied theoretical foundations [13, 15, 26, 27], which we have also applied in this paper. Algebraic effects can also be used in Haskell [29], though in a less powerful way than described here. 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. Nevertheless, since `EffM` is represented as an algebraic data type, some effect inference would be possible.

While our original motivation was to avoid the need for monad transformers [16] in order to compose effects, there is a clear relationship. Indeed, `EffM` 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 [31] or Haskell extensions [18], 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, 28], which similarly 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, 20].

To some extent, we can now support imperative programming with dependent types, such as supported by Xanadu [32] and Ynot [22]. Ynot in particular is an axiomatic extension to Coq which allows reasoning about imperative programs using Hoare Type Theory [21] — preconditions and postconditions on operations can be expressed in `Effects` by giving appropriate 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 old 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 representable 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 `IO` 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

[2] `https://personal.cis.strath.ac.uk/conor.mcbride/pub/Frank/`

[3] `http://research.microsoft.com/en-us/projects/koka/`

realistic programs. The most obvious weakness of `Effects`, which is already known for the algebraic approach, 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 `EffM` 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.

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 — no extensions were 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 sub-list 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.

References

[1] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015—-1012, 2009.

[2] L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*. Citeseer, 1999.

[3] A. Bauer and M. Pretnar. Programming with Algebraic Effects and Handlers, 2012.

[4] E. Brady. Idris, a General Purpose Programming Language : Design and Implementation, 2013.

[5] E. Brady. Programming in Idris : a tutorial, 2013.

[6] E. Brady and K. Hammond. A verified staged interpreter is a verified compiler. In *Generative Programming and Component Engineering (GPCE 2006)*, 2006.

[7] E. Brady and K. Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102, 2010.

[8] E. Brady and K. Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 297–308. ACM, 2010.

[9] E. Brady and K. Hammond. Resource-safe Systems Programming with Embedded Domain Specific Languages. In *Practical Applications of Declarative Languages*, pages 242—-257, 2012.

[10] J. Carette, O. Kiselyov, and C.-C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543.

[11] D. Delahaye. A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning (LPAR)*, pages 85—-95. Springer, 2000.

[12] P. Hancock and A. Setzer. Interactive Programs in Dependent Type Theory. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, pages 317–331, Aug. 2000.

[13] M. Hyland, G. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 2006.

[14] M. Jaskelioff. Modular monad transformers. In *ESOP 09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 64—-79, 2009.

[15] P. B. Levy. *Call-By-Push-Value*. PhD thesis, Queen Mary and Westfield College, University of London, 2001.

[16] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 333–343, New York, New York, USA, Jan. 1995. ACM Press.

[17] B. Lippmeier. Witnessing Purity, Constancy and Mutability. In *7th Asian Symposium, (APLAS 2009)*, pages 95—-110, 2009.

[18] C. McBride. Kleisli arrows of outrageous fortune, 2011.

[19] C. McBride and R. Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1—-13, 2008.

[20] M. Michelbrink and A. Setzer. State dependent IO-monads in type theory. *Electronic Notes in Theoretical Computer Science*, 122:127—-146, 2005.

[21] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5–6):865—-911, Sept. 2008.

[22] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming*, number 9, pages 229—-240, 2008.

[23] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, 2004.

[24] E. Pašalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *International Conference on Functional Programming*, volume 37, pages 218–229, Sept. 2002.

[25] S. Peyton Jones. Haskell 98 Language and Libraries: The Revised Report. Technical Report 1, 2002.

[26] G. Plotkin and M. Pretnar. Handlers of Algebraic Effects. In *ESOP 09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 80—-94, 2009.

[27] M. Pretnar. *The Logic and Handling of Algebraic Effects*. PhD thesis, University of Edinburgh, 2010.

[28] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157—-171, 1986.

[29] W. Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423—-436, 2008.

[30] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science, pages 24—-52. Springer, 1995.

[31] D. Walker. A Type System for Expressive Security Policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 254–267. ACM, 2000.

[32] H. Xi. Imperative Programming with Dependent Types. In *15th Annual IEEE Symposium on Logic in Computer Science*, pages 375—-387, 2000.