# Resource-dependent Algebraic Effects

Edwin Brady

University of St Andrews, KY16 9SX, Scotland/UK,
ecb10@st-andrews.ac.uk

**Abstract** There has been significant interest in recent months in finding new ways to implement composable and modular effectful programs using *handlers of algebraic effects*. In my own previous work, I have shown how an algebraic effect system (called `effects`) can be embedded directly in a dependently typed host language. Using dependent types ought to allow precise reasoning about programs; however, the reasoning capabilities of `effects` have been limited to simple state transitions which are known at compile-time. In this paper, I show how `effects` can be extended to support reasoning in the presence of *run-time* state transitions, where the result may depend on run-time information about resource usage (e.g. whether opening a file succeeded). I show how this can be used to build expressive APIs, and to specify and verify the behaviour of interactive, stateful programs. I illustrate the technique using a file handling API, and an interactive game.

## 1 Introduction

Pure functional languages with dependent types such as Idris [3] support reasoning about programs directly in the type system, promising that we can *know* a program will run correctly (i.e. according to the specification in its type) simply because it compiles. However, things are not always so simple: programs have to interact with the outside world, with user input, input from a network or mutable state. Such operations are outside the control of a language, and may fail.

In previous work [4], I showed how Idris could be used to manage stateful and side-effecting programs in an Embedded Domain Specific Language (EDSL) called `effects`, built around an implementation of algebraic effects and handlers [2,15]. Informally, an algebraic effect is an algebraic datatype describing a collection of permitted operations. For example, the `STDIO` effect for Console I/O supports the operations `getStr` and `putStr` and the `EXCEPTION` effect supports the operation `raise`. The `effects` EDSL allows us to *compose* effects in one program, e.g.

```
readName : List String ->
             { [STDIO, EXCEPTION String] } Eff ()
readName known
   = do putStr "Name: "
        x <- getStr
        if (trim x `elem` known)
```

```
            then putStr $ "Hello " ++ x ++ "\n"
            else raise "Name not recognised"
```

This program reads a name from the console, and prints a different message depending on whether the user's name is recognised or not. The effects `STDIO` and `EXCEPTION String` are given in the type to express that the program supports Console I/O and exceptions which carry strings, respectively. Effectful programs are executed with the `run` function, e.g.:

```
main : IO ()
main = run (readName ["Alice","Bob"])
```

Being implemented in a dependently typed language, `effects` supports reasoning about state transitions. The previous implementation was, however, seriously limited in that only known compile-time transitions could be expressed. Opening a file, for example, was assumed to always succeed with any failure being dealt with in an exception handler. Realistically, any interaction with the outside world is likely to fail: files may not open, network transmissions may fail, users may input invalid data. We would like to be able to state precisely *how* effectful operations may affect the state of the outside world, what failures might occur, and guarantee that they are all handled appropriately, ideally without imposing significant proof burden on a progammer.

### 1.1 Contributions

In the rest of this paper I describe, by example, a more sophisticated implementation of `effects` which supports reasoning about state transitions which are not known until run-time, e.g. whether opening a file was successful, overcoming the previous limitations. I make the following specific contributions:

- I show how parameterising algebraic effects over resources leads to the ability to reason about stateful, side-effecting programs.
- I show how `effects`, when extended to support parameterised effects, can support precise run-time dependent APIs for stateful libraries.
- I give a concrete example of a stateful program, a mystery word guessing game, which is specified as a resource-dependent algebraic effect.

## 2 Effectful Programming in Idris

In this section, I give a brief introduction to programming with side-effects in Idris. A complete tutorial[1] and details of the implementation [4] are given elsewhere.

An effectful program `f` has a type of the following form:

```
f : (x1 : a1) -> (x2 : a2) -> ... ->
    { eff ==> {result} effs' } Eff t
```

[1] `http://eb.host.cs.st-andrews.ac.uk/drafts/eff-tutorial.pdf`

That is, the return type gives the effects that `f` supports (`effs`, of type `List EFFECT`), the effects available *after* running `f` (`effs'`) which may be calculated using the result of the operation `result` of type `t`.

A function which does not update its available effects has a type of the following form:

```
f : (x1 : a1) -> (x2 : a2) -> ... -> { eff } Eff m t
```

In fact, the notation `{ eff }` is itself syntactic sugar, in order to make `Eff` types more readable. In full, the type of `Eff` is:

```
Eff : (x : Type) ->
      List EFFECT -> (x -> List EFFECT) -> Type
```

That is, it is indexed over the type of the computation, the list of input effects and a function which computes the output effects from the result. IDRIS supports a notation for extending syntax, which allows us to create syntactic sugar for `Eff` as described above:

```
syntax "{" [inst] "}" [eff] = eff inst (\result => inst)
syntax "{" [inst] "==>" "{" {b} "}" [outst] "}" [eff]
       = eff inst (\b => outst)
syntax "{" [inst] "==>" [outst] "}" [eff]
       = eff inst (\result => outst)
```

### 2.1 Example Effectful Programs

A program which carries a state and outputs it to the console would have the following type:

```
writeState : Show a => { [STATE a, STDIO] } Eff ()
```

That is, it can read and write a state of type `a` and it can perform Console I/O. Each effect in the given list carries a corresponding *resource* which is used when executing an effectful program. `STATE a` for example carries a resource of type `a`. If there are multiple effects of the same type (for example, multiple states), they can be disambiguated by *labelling*, although we will not require this in the present paper.

More generally, a function can *update* the available effects, depending on its output. For example, a program which attempts to open a file in a particular mode (`Read` or `Write`) has the following type:

```
open : String -> (m : Mode) ->
     { [FILE_IO ()] ==>
       {ok} [FILE_IO (if ok then OpenFile m else ())] }
       Eff Bool
```

The `FILE_IO` effect carries the current state of a file handle. It begins as the unit type (i.e. no file handle is carried in its resource). If opening the file is successful

(i.e., `open` returns `True` and hence `ok` is `True`) then a file handle is available, otherwise it is not.

If a file is available which is open for reading, we can use `readFile` to retrieve its contents:

```
readFile : { [FILE_IO (OpenFile Read)] }
           Eff (List String)
```

Using this, we can write a program which opens a file, reads it, then displays the contents and closes it, correctly following a resource usage protocol (where the `!`-notation, directly applying an effectful operation, is explained further below):

```
dumpFile : String -> { [FILE_IO (), STDIO] } Eff ()
dumpFile name = case !(open name Read) of
                     True => do putStrLn (show !readFile)
                                close
                     False => putStrLn ("Error!")
```

The type of `dumpFile`, with `FILE_IO ()` in its effect list, indicates that any use of the file resource will follow the protocol correctly (i.e. it both begins and ends with an empty resource). If we fail to follow the protocol correctly (perhaps by forgetting to close the file, failing to check that `open` succeeded, or opening the file for writing) then we will get a compile-time error.

### 2.2 !-notation

Just as with monadic programming in Haskell, we can use `do`-notation to sequence effectful operations. In many cases, however, `do`-notation can make programs unnecessarily verbose, particularly in cases where the value bound is used once, immediately. The following program returns the length of the `String` stored in a state, for example:

```
stateLength : { [STATE String] } Eff m Nat
stateLength = do x <- get
                 pure (length x)
```

Here, `pure` injects a pure value into an effectful program (like `return` in Haskell). Idris provides `!`-notation to allow a more direct style:

```
stateLength : { [STATE String] } Eff m Nat
stateLength = pure (length !get)
```

The notation `!expr` means that the expression `expr` should be evaluated and then implicitly bound. Conceptually, we can think of `!` as being a prefix function with the following type:

```
(!) : { xs } Eff m a -> a
```

Note, however, that it is *syntax*, not a function. Indeed, such a function would be impossible to implement in general. On encountering a subexpression `!expr`, Idris will lift `expr` out as far as possible within its current scope, bind it to

a fresh name `x`, and replace `!expr` with `x`. Expressions are lifted depth first, left to right. In practice, `!`-notation allows us to program in a more direct style, while still giving a notational clue as to which expressions are effectful.

For example, the expression...

```
let y = 42 in f !(g !(print y) !x)
```

...is lifted to:

```
let y = 42 in do printy' <- print y
                 x' <- x
                 g' <- g printy' x'
                 f g'
```

### 2.3 Pattern-matching bind

It might seem that having to test each potentially failing operation with a `case` clause could lead to ugly code, with lots of nested case blocks. Many languages support exceptions to improve this, but unfortunately exceptions may not allow completely clean resource management. For example, guaranteeing that any successful `open` has a corresponding `close` becomes difficult when an exception could be thrown in between the operations.

IDRIS supports *pattern-matching* bindings, such as the following:

```
dumpFile : String -> { [FILE_IO (), STDIO] } Eff ()
dumpFile name = do True <- open name Read
                   putStrLn (show !readFile)
                   close
```

This also has a problem: we are no longer dealing with the case where opening a file failed! The IDRIS solution is to extend the pattern-matching binding syntax to give clauses for failing matches. Here, for example, we could write:

```
dumpFile : String -> { [FILE_IO (), STDIO] } Eff ()
dumpFile name  = do True <- open name Read
                                | False => putStrLn "Error"
                    putStrLn (show !readFile)
                    close
```

This is exactly equivalent to the definition with the explicit `case`. In general, in a `do`-block, the syntax...

```
do pat <- val | <alternatives>
   p
```

...is desugared to...

```
do x <- val
   case x of
        pat => p
        <alternatives>
```

There can be several `alternatives`, separated by a vertical bar `|`.

## 3 Implementing Resource-Dependent Effects

In this section, I show how effects can be implemented in order to model resource usage protocols, using the `STATE` effect as an illustrative example of simple effects, and extending this to resource-dependent effects with error handling, using `FILE_IO`.

### 3.1 State

Effects are described by *algebraic data types*, where the constructors describe the operations provided by the effect. Stateful operations are described as follows:

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

Each effect is associated with a *resource*, the type of which is given with the notation `{ x ==> x' }`. This notation gives the resource type expected by each operation, and how it updates when the operation is run. Here, it means:

- `Get` takes no arguments. It has a resource of type `a`, which is not updated, and running the `Get` operation returns something of type `a`.
- `Put` takes a `b` as an argument. It has a resource of type `a` on input, which is updated to a resource of type `b`. Running the `Put` operation returns the element of the unit type.

`Effect` itself is a type synonym. In Idris, type synonyms are simply functions, since functions can compute types. It is declared as follows:

```
Effect : Type
Effect = (result : Type) ->
         (input_resource : Type) ->
         (output_resource : result -> Type) -> Type
```

That is, an effectful operation returns something of type `result`, has an input resource of type `input_resource`, and a function `output_resource` which computes the output resource type from the result. We use the same syntactic sugar as with `Eff` to make effect declarations more readable, and specifically to make the state transition clear.

In order to convert `State` (of type `Effect`) into something usable in an effects list, of type `EFFECT`, we write the following:

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

`MkEff` constructs an `EFFECT` by taking the resource type (here, the `t` which parameterises `STATE`) and the effect signature (here, `State`). For reference, `EFFECT` is declared as follows:

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

To be able to `run` an effectful program in `Eff`, we must explain how it is executed. Programs are run in some *computation context* which supports the underlying effects (e.g. console I/O runs under `IO`). Instances of the following class describe how an effect is executed in a particular context:

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

An instance of `Handler e m` means that the effect declared with signature `e` can be run in computation context `m`. The name `m` is suggestive of a monad, although there is no requirement for it to be so. For example, the identity function `id` would be valid for effects which can execute in a pure context. The `handle` function takes:

- The resource `res` on input (so, the current value of the state for `State`)
- The effectful operation (either `Get` or `Put x` for `State`)
- A *continuation*, which we conventionally call `k`, and should be passed the result value of the operation, and an updated resource.

A `Handler` for `State` simply passes on the value of the state, in the case of `Get`, or passes on a new state, in the case of `Put`. It is defined the same way for all computation contexts:

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

This gives enough information for `Get` and `Put` to be used directly in `Eff` programs. It is tidy, however, to define top level functions in `Eff`, as follows:

```
get : { [STATE x] } Eff x
get = call Get

put : x -> { [STATE x] } Eff ()
put val = call (Put val)

putM : y -> { [STATE x] ==> [STATE y] } Eff ()
putM val = call (Put val)
```

The `call` function converts an `Effect` to a function in `Eff`, given a proof that the effect is available. This proof can be constructed automatically by Idris, since it is essentially an index into a statically known list of effects:

```
call : {e : Effect} ->
       (eff : e t a b) -> {auto prf : EffElem e a xs} ->
       Eff t xs (\v => updateResTy v xs prf eff)
```

### 3.2 File Management

Result-dependent effects are, in general, no different from non-dependent effects in the way they are implemented, other than the transitions being made explicit in the declaration.. The `FILE_IO` effect, for example, is declared as in Listing 1.

**Listing 1.** File I/O effect

```
data FileIO : Effect where
     Open  : String -> (m : Mode) ->
             {() ==> {ok} if ok
                             then OpenFile m
                             else ()} FileIO Bool
     Close : {OpenFile m ==> ()}       FileIO ()

     ReadLine  :           {OpenFile Read}  FileIO String
     WriteLine : String -> {OpenFile Write} FileIO ()
     EOF       :           {OpenFile Read}  FileIO Bool
```

The syntax for state transitions `{ x ==> {res} x' }`, where the result state `x'` is computed from the result of the operation `res`, follows that for the equivalent `Eff` programs. The distinctive operation declared in this effect signature is `Open`, the type of which captures the possibility of failure.

Before executing `Open`, the resource state must be empty (i.e., there is no file handle). After executing `Open`, we either have a file handle, open for the appropriate mode (if `ok` is `True`) or no file. This can be made into a function in `Eff` as follows (we have already seen the type of `open` in Section 2.1):

```
open : String -> (m : Mode) ->
     { [FILE_IO ()] ==>
       {ok} [FILE_IO (if ok then OpenFile m else ())] }
       Eff Bool
open f m = Open f m
```

This type illustrates the crucial distinction between resource-dependent effects and the previous implementation [4]. Namely, the output effects are *computed* for a result which will become known only at run-time. As a result, the only way for a program using the `open` operation to be well-typed is for it to *check* the result at run-time:

```
dumpFile : String -> { [FILE_IO (), STDIO] } Eff ()
dumpFile name = case !(open name Read) of
                     True => do putStrLn (show !readFile)
                                close
                     False => putStrLn ("Error!")
```

By performing `case` analysis on the result of `open name Read`, the type of the resource in each branch is specialised according to whether the result is `True` or

`False`, meaning that the `if...then...else` construct in the output resource can be reduced further. The `Handler` for `FileIO` is written as in Listing 2.

**Listing 2.** File I/O handler

```
instance Handler FileIO IO where
    handle () (Open fname m) k
           = do h <- openFile fname m
                if !(validFile h) then k True (FH h)
                                  else k False ()
    handle (FH h) Close k
           = do closeFile h
                k () ()

    handle (FH h) ReadLine         k = do str <- fread h
                                          k str (FH h)
    handle (FH h) (WriteLine str) k = do fwrite h str
                                          k () (FH h)
    handle (FH h) EOF              k = do e <- feof h
                                          k e (FH h)
```

Note that in the handler for `Open`, the types passed to the continuation `k` are different depending on whether the result is `True` (opening succeeded) or `False` (opening failed). This uses `validFile`, defined in the `Prelude`, to test whether a file handler refers to an open file or not.

## 4 Example: A "Mystery Word" Guessing Game

In this section, we will use `effects` to implement a larger example, a simple text-based word-guessing game. The effect will allow us to express the rules of the game formally and precisely, with a resource-dependent effect allowing the machine to update game state at run-time, according to information which will only be known at run-time.

In the game, the computer chooses a word, which the player must guess letter by letter. The player wins when all the letters have been guessed correctly, and loses after a limited number of wrong guesses[2]. We will implement the game by following these steps:

1. Define the game state, in enough detail to express the rules
2. Define the rules of the game (i.e. what actions the player may take, and how these actions affect the game state)
3. Implement the rules of the game (i.e. implement state updates for each action)
4. Implement a user interface which allows a player to direct actions

[2] Readers may recognise this game by the name "Hangman".

Step 2 may be achieved by defining an effect which depends on the state defined in Step 1. Then Step 3 involves implementing a `Handler` for this effect. Finally, Step 4 involves implementing a program in `Eff` using the newly defined effect (and any others required to implement the interface). By using `effects`, we can be certain that our implementation of the game follows the rules we have specified.

### 4.1 Step 1: Game State

First, we categorise the game states as running games (where there are a number of guesses available, and a number of letters still to guess), or non-running games (i.e. games which have not been started, or games which have been won or lost).

```
data GState = Running Nat Nat | NotRunning
```

Notice that at this stage, we say nothing about what it means to make a guess, what the word to be guessed is, how to guess letters, or any other implementation detail. We are only interested in what is necessary to describe the game rules. We will, however, parameterise a concrete game state `Mystery` over this data:

```
data Mystery : GState -> Type
```

### 4.2 Step 2: Game Rules

We describe the game rules as a resource-dependent effect, where each action has a *precondition* (i.e. what the game state must be before carrying out the action) and a *postcondition* (i.e. how the action affects the game state). Informally, these actions with the pre- and postconditions are:

**Guess** Guess a letter in the word.
- Precondition: The game must be running, and there must be both guesses still available, and letters still to be guessed.
- Postcondition: If the guessed letter is in the word and not yet guessed, reduce the number of letters, otherwise reduce the number of guesses.

**Won** Declare victory
- Precondition: The game must be running, with no letters still to be guessed.
- Postcondition: The game is no longer running.

**Lost** Accept defeat
- Precondition: The game must be running, with no guesses left.
- Postcondition: The game is no longer running.

**NewWord** Set a new word to be guessed
- Precondition: The game must not be running.
- Postcondition: The game is running, with 6 guesses available (the choice of 6 is somewhat arbitrary here) and the number of unique letters in the word still to be guessed.

**StrState** Get a string representation of the game state. This is for display purposes; there are no pre- or postconditions.

We can make these rules precise by declaring them in an effect signature:

```
data MysteryRules : Effect where
  Guess : (x : Char) ->
          { Mystery (Running (S g) (S w)) ==>
            {inword} if inword
                        then Mystery (Running (S g) w)
                        else Mystery (Running g (S w)) }
           MysteryRules Bool
  Won  : { Mystery (Running g 0) ==>
           Mystery NotRunning } MysteryRules ()
  Lost : { Mystery (Running 0 g) ==>
           Mystery NotRunning } MysteryRules ()
  NewWord : (w : String) ->
            { Mystery NotRunning ==>
              Mystery (Running 6 (length (letters w))) }
            MysteryRules ()
  StrState : { Mystery h } MysteryRules String
```

This description says nothing about how the rules are implemented. In particular, it does not specify *how* to tell whether a guessed letter was in a word, just that the result of `Guess` depends on it.

Nevertheless, we can still create an `EFFECT` from this, and use it in an `Eff` program. Implementing a `Handler` for `MysteryRules` will then allow us to play the game by filling in the missing implementation details.

```
MYSTERY : GState -> EFFECT
MYSTERY h = MkEff (Mystery h) MysteryRules
```

### 4.3 Step 3: Implement Rules

To *implement* the rules, we begin by giving a concrete definition of game state:

```
data Mystery : GState -> Type where
     Init     : Mystery NotRunning
     GameWon  : (word : String) -> Mystery NotRunning
     GameLost : (word : String) -> Mystery NotRunning
     MkG      : (word : String) -> (guesses : Nat) ->
                (got : List Char) ->
                (missing : Vect m Char) ->
                Mystery (Running guesses m)
```

If a game is `NotRunning`, that is either because it has not yet started (`Init`) or because it is won or lost (`GameWon` and `GameLost`, each of which carry the word so that showing the game state will reveal the word to the player). Finally,

`MkG` captures a running game's state, including the target word, the letters successfully guessed, and the missing letters. Using a `Vect` for the missing letters is convenient since its length is used in the type of `Mystery` itself. This makes the link between the missing letters and the game state explicit, and statically checkable.

To initialise the state, we implement the following functions: `letters`, which returns a list of unique letters in a `String` (ignoring spaces) and `initState` which sets up an initial state considered valid as a postcondition for `NewWord`.

```
letters : String -> List Char
initState : (x : String) ->
            Mystery (Running 6 (length (letters x)))
```

When checking if a guess is in the vector of missing letters, it is convenient to return a *proof* that the guess is in the vector, using `isElem` below, rather than merely a `Bool`. This is defined in the IDRIS prelude:

```
data IsElem : a -> Vect n a -> Type where
     First : IsElem x (x :: xs)
     Later : IsElem x xs -> IsElem x (y :: xs)

isElem : DecEq a =>
    (x : a) -> (xs : Vect n a) -> Maybe (IsElem x xs)
```

The reason for returning a proof is that we can use it to remove an element from the correct position in a vector:

```
shrink : (xs : Vect (S n) a) -> IsElem x xs -> Vect n a
```

Having implemented these, the `Handler` implementation for `MysteryRules` simply involves directly updating the game state in a way which is consistent with the declared rules:

```
instance Handler MysteryRules m where
    handle (MkG w g got []) Won k = k () (GameWon w)
    handle (MkG w Z got m) Lost k = k () (GameLost w)

    handle st StrState k = k (show st) st
    handle st (NewWord w) k = k () (initState w)

    handle (MkG w (S g) got m) (Guess x) k =
        case isElem x m of
             Nothing => k False (MkG w _ got m)
             (Just p) =>
                k True (MkG w _ (x :: got) (shrink m p))
```

In particular, in `Guess`, if the handler claims that the guessed letter is in the word (by passing `True` to `k`), there is no way to update the state in such a way that the number of missing letters or number of guesses does not follow the rules. This would be a *compile-time* type error, due to the link between the game state's type and the vector of missing letters.

### 4.4 Step 4: Implement Interface

Having described the rules, and implemented state transitions which follow those rules as an effect handler, we can now write an interface for the game which uses the MYSTERY effect:

```
game : { [MYSTERY (Running (S g) w), STDIO] ==>
         [MYSTERY NotRunning, STDIO] } Eff ()
```

The type indicates that the game must start in a running state, with some guesses available, and eventually reach a not-running state (i.e. won or lost). The only way to achieve this is by correctly following the stated rules. A possible complete implementation of game is presented in Listing 3.

Note that the type of game makes no assumption that there are letters to be guessed in the given word (i.e. it is w rather than S w). This is because we will be choosing a word at random from a vector of Strings, and at no point have we made it explicit that those Strings are non-empty.

Finally, we need to initialise the game by picking a word at random from a list of candidates, setting it as the target using NewWord, then running game:

```
runGame : { [MYSTERY NotRunning, RND, SYSTEM, STDIO] }
          Eff ()
runGame = do srand (cast !time)
             let w = index !(rndFin _) words
             NewWord w
             game
             putStrLn !StrState
```

We use the system time (provided by SYSTEM) to initialise the random number generator (provided by RND), then pick a random element of a finite set Fin to index into a list of words. For example, we could initialise a word list as follows:

```
words : ?wtype
words = with Vect ["idris","agda","haskell","miranda",
         "java","javascript","fortran","basic","erlang",
         "racket","clean","links","coffeescript","rust"]

wtype = proof search
```

**Aside:** Rather than have to explicitly declare a type with the vector's length, it is convenient to give a metavariable ?wtype and let Idris's proof search mechanism find the type. This is a limited form of type inference, but very useful in practice.

## 5 Related Work

There has been much recent interest in using algebraic effects to support moduler, composable effectful programming. The effects library was initially inspired by Bauer and Pretnar's Eff language [2], and there have been successful efforts

**Listing 3.** Mystery Word Game Implementation

```
game : { [MYSTERY (Running (S g) w), STDIO] ==>
         [MYSTERY NotRunning, STDIO] } Eff ()
game {w=Z} = Won
game {w=S _}
     = do putStrLn !StrState
          putStr "Enter guess: "
          let guess = trim !getStr
          case choose (not (guess == "")) of
               (Left p) => processGuess (strHead' guess p)
               (Right p) => do putStrLn "Invalid input!"
                               game
  where
    processGuess : Char ->
                   { [MYSTERY (Running (S g) (S w)), STDIO] ==>
                     [MYSTERY NotRunning, STDIO] }
                   Eff ()
    processGuess {g} {w} c
      = case !(Main.Guess c) of
             True => do putStrLn "Good guess!"
                        case w of
                             Z => Won
                             (S k) => game
             False => do putStrLn "No, sorry"
                         case g of
                              Z => Lost
                              (S k) => game
```

to implement handlers of algebraic effects in Haskell and other languages [9,10]. Unlike the `effects` library, these systems do not attempt to support reasoning about resource usage or state updates, but are flexible in other ways such as allowing handlers of effects to be reordered. Other languages aim to bring effects into their type system, such as Disciple [13], Koka [11] and Frank[3]. These languages are built on well-studied theoretical foundations [8,12,15,16], which we have also applied in the `effects` library.

The resource-dependent effect library described in this paper is a refinement of previous work [4] implementing algebraic effects in Idris. An important limitation of this work was the difficulty of dealing with errors. This was improved to some extent in order to implement libraries for web programming [6] by adding an explicit error-checking construct, but this too has proved limited in practice for implementing more complex protocols. Inspired by McBride [14], the present implementation allows the result type of an effectful operation to depend on run-time information, with *compile-time* checks enforced by the type system ensuring that any necessary *run-time* checks are made.

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

The problem of reasoning about resource usage protocols has previously been tackled using special purpose type systems [18], by creating DSLs for resource management [5], or with Typestate [1,17]. These are less flexible than the `effects` approach, however, since combining resources is difficult. In `effects`, we can combine resources simply by extending the list of available effects.

## 6 Conclusion

The `effects` system extends the previous implementation by allowing precise reasoning about state updates, even in the presence of information which is not known until run-time. By capturing the possibility of failure in the resource state of an effect, we know that a programmer cannot avoid handling failure. Lightweight syntactic sugar, such as `!`-notation and pattern matching alternatives mean that programs remain short and readable.

In the Mystery Word game, I wrote the rules separately as an effect, then wrote an implementation which uses that effect. This ensures that the implementation must follow the rules. In practice, we would not expect to follow a strict process of writing the rules first then implementing the game once the rules were complete. Indeed, I did not do so when constructing the example! Rather, I wrote down a first draft of the rules making any assumptions *explicit* in the state transitions for `MysteryRules`. Then, when implementing `game` at first, any incorrect assumption was caught as a type error. The following errors were caught during development:

- Not realising that allowing `NewWord` to be an arbitrary string would mean that `game` would have to deal with a zero-length word as a starting state.
- Forgetting to check whether a game was won before recursively calling `processGuess`, thus accidentally continuing a finished game.
- Accidentally checking the number of missing letters, rather than the number of remaining guesses, when checking if a game was lost.

While these are simple errors, they were caught by the type checker before *any* testing of the game. This approach has practical applications in more serious contexts; `MysteryRules` for example can be thought of as describing a *protocol* that a game player most follow, or alternative a *precisely-typed API*. Precise reasoning about resource usage, and constraints on ordering of operations and error checking, can be particularly important in safety and security critical contexts. For example, a recent security flaw in Apple's iOS[4] was caused in part by faulty error handling code in an SSL key exchange protocol.

We are currently using resource-dependent effects to implement a DSL for type-safe communication, similar to session types [7]. Using this, we plan to investigate verification of security properties of protocols. In this context, resource-dependency is essential: the execution of the protocol depends on values which are communicated across a network or given by a user, which cannot be known until run-time.

[4] `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266`

## References


1. J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of the 24th conference on Object Oriented Programming Systems Languages and Applications*, pages 1015–1012, 2009.
2. A. Bauer and M. Pretnar. Programming with Algebraic Effects and Handlers, 2012. Available from `http://arxiv.org/abs/1203.1539`.
3. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.
4. E. Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. In *ICFP '13: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2013.
5. E. Brady and K. Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102:145–176, 2010.
6. S. Fowler and E. Brady. Dependent types for safe and secure web programming. In *Implementation and Application of Functional Languages (IFL 2013)*, 2013. To appear.
7. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.
8. M. Hyland, G. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357:70–99, 2006.
9. O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th International Conference on Functional Programming (ICFP '13)*. ACM, 2013.
10. O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 59–70, New York, NY, USA, 2013. ACM.
11. D. Leijen. Koka: Programming with row polymorphic effect types. In P. Levy and N. Krishnaswami, editors, *MSFP*, volume 153 of *EPTCS*, pages 100–126, 2014.
12. P. B. Levy. *Call-By-Push-Value*. PhD thesis, Queen Mary and Westfield College, University of London, 2001.
13. B. Lippmeier. Witnessing Purity, Constancy and Mutability. In *7th Asian Symposium on Programming Languages and Systems (APLAS 2009)*, volume 5904 of *LNCS*, pages 95–110. Springer, 2009.
14. C. McBride. Kleisli arrows of outrageous fortune, 2011. Draft.
15. 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.
16. M. Pretnar. *The Logic and Handling of Algebraic Effects*. PhD thesis, University of Edinburgh, 2010.
17. 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.
18. 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.